L Number	Hits	Search Text	DB	Time stamp
1	389	717/150.ccls. 717/159-161.ccls.	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 11:44
4	6569	prefetch pre adj fetch	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 11:50
7	27	cache adj management adj instruction	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:50
9	1904	(detail section) adj loop	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 11:59
10	590	loop near5 (unroll$3 un adj roll$3 restructur$3)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 12:00
11	51	(loop near5 (unroll$3 un adj roll$3 restructur$3)) and (prefetch pre adj fetch)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 12:02
13	4	((loop near5 (unroll$3 un adj roll$3 restructur$3)) and (prefetch pre adj fetch)) and ((detail section) adj loop)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:37
15	10	(prefetch pre adj fetch) and (cache adj management adj instruction)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:44
16	2	((prefetch pre adj fetch) and (cache adj management adj instruction)) and (loop near5 (unroll$3 un adj roll$3 restructur$3))	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:47
17	3872354	code loop source object	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:48
18	9	((prefetch pre adj fetch) and (cache adj management adj instruction)) and (code loop source object)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:49
20	156	((prefetch pre adj fetch cache adj management) adj instruction) near3 (source code loop)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:52
21	32	((prefetch pre adj fetch cache adj management) adj instruction) near3 (loop)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2004/01/29 13:52

US005950007A

United States Patent [19]

Nishiyama et al.

[11] **Patent Number: 5,950,007**

[45] **Date of Patent: Sep. 7, 1999**

[54] **METHOD FOR COMPILING LOOPS CONTAINING PREFETCH INSTRUCTIONS THAT REPLACES ONE OR MORE ACTUAL PREFETCHES WITH ONE VIRTUAL PREFETCH PRIOR TO LOOP SCHEDULING AND UNROLLING**

[75] Inventors: **Hiroyasu Nishiyama**, Kawasaki; **Sumio Kikuchi**, Machida, both of Japan

[73] Assignee: **Hitachi, Ltd.**, Tokyo, Japan

[21] Appl. No.: **08/675,964**

[22] Filed: **Jul. 5, 1996**

[30] **Foreign Application Priority Data**

Jul. 6, 1995 [JP] Japan 7-170674

[51] **Int. Cl.**^{6} **G06F 9/45**

[52] **U.S. Cl.** **395/707**; 395/705

[58] **Field of Search** 395/705, 706, 395/707, 709, 710

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,303,357	4/1994	Inoue et al.	395/709
5,367,651	11/1994	Smith et al.	395/709
5,491,823	2/1996	Ruttenberg	395/709
5,557,761	9/1996	Chan et al.	395/709
5,664,193	9/1997	Tirumalai	395/705
5,704,053	12/1997	Santhanam	395/383
5,752,037	5/1998	Gornish et al.	395/709
5,761,515	6/1998	Barton, III et al.	395/709
5,794,029	8/1998	Babaian et al.	395/588
5,797,013	8/1998	Mahadevan et al.	395/709
5,809,308	9/1998	Tirumalai	395/709
5,819,088	10/1998	Reinders	395/672
5,835,776	11/1998	Tirumalai et al.	395/709

OTHER PUBLICATIONS

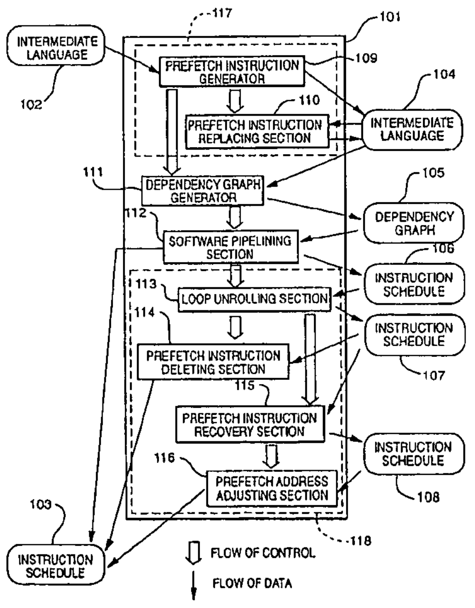
S. Ramakrishnan, "Software Pipelining in PA–RISC Compilers," Hewlett–Packard Journal, Jun. 1992, pp. 39–45.
T. Mowry et al, "Design and Evaluation of a Compiler Algorithm for Prefetching," Proceedings of the Fifth International Conference on Architectural Support for Programming Language and Operating Systems, 1992, pp. 62–73.
D. Callahan et al, "Software Prefetching," 1991 ACM 0–89791–380, pp. 40–52.
G. Kurpanek et al, "PA7200: A PA–RISC Processor with Integrated High Performance MP Bus Interface," Hewlett–Packard Company, 1994 IEEE, pp. 375–382.

Primary Examiner—Zarni Maung
Assistant Examiner—Andrew Caldwell
Attorney, Agent, or Firm—Fay, Sharpe, Beall, Fagan, Minnich & McKee

[57] **ABSTRACT**

Prefetch instructions having a function to move data to a cache memory from main memory are scheduled simultaneously with execution of other instructions. The prefetch instructions are scheduled by replacing, with the original prefetch instructions, the virtual prefetch instructions obtained by unrolling a kernel section of the schedule constituted by generating a dependency graph having dependent relationships between the prefetch instruction and the memory reference instruction, and then applying the software pipelining thereto, or by further unrolling the kernel section of the constituted schedule to delete the redundant prefetch instructions, or further by applying the software pipelining to the dependency graph which is formed by combining a plurality of prefetch instructions and replacing the prefetch instructions with virtual prefetch instructions.

6 Claims, 16 Drawing Sheets



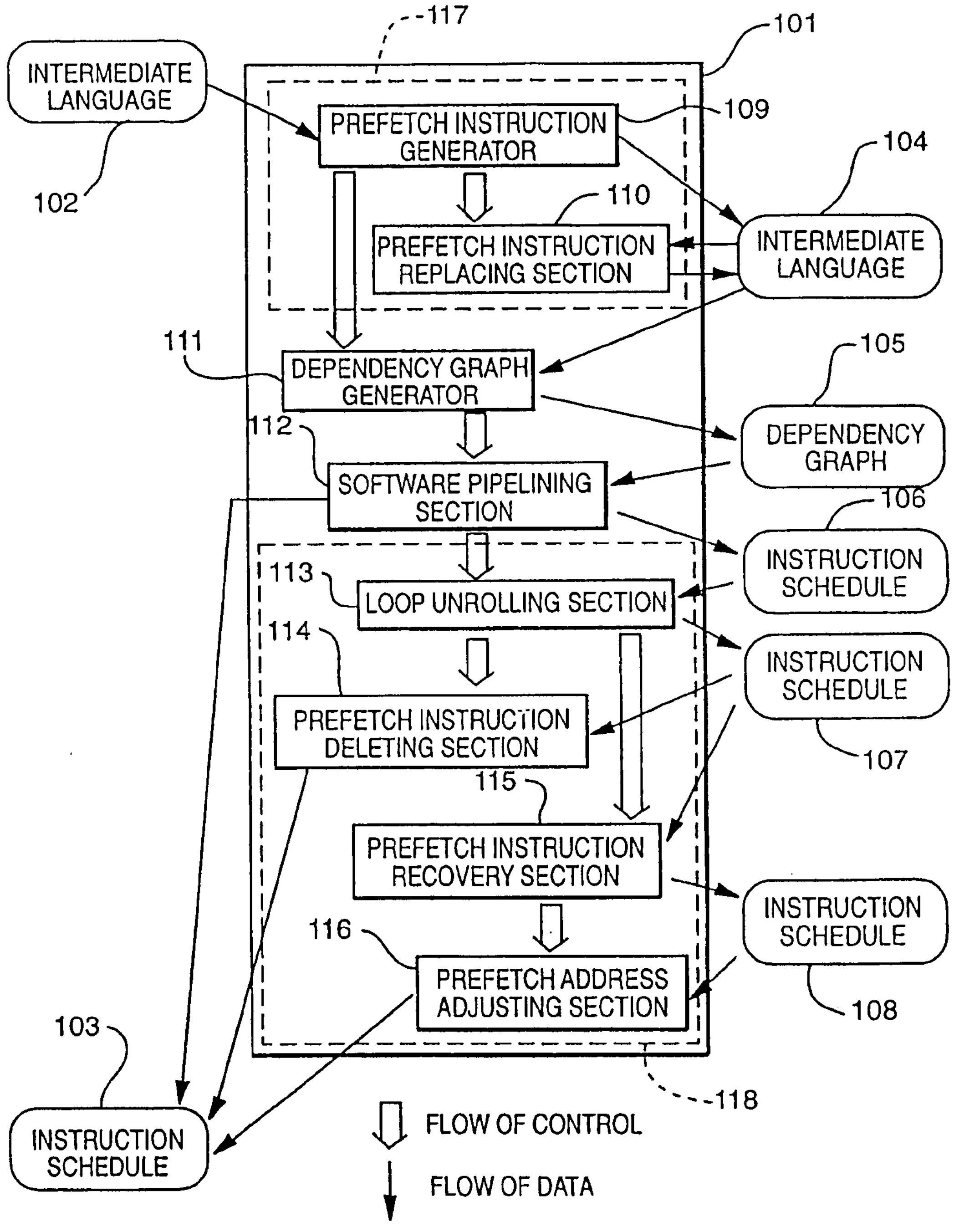

117
101
INTERMEDIATE LANGUAGE
102
PREFETCH INSTRUCTION GENERATOR
109
104
110
PREFETCH INSTRUCTION REPLACING SECTION
INTERMEDIATE LANGUAGE
111
DEPENDENCY GRAPH GENERATOR
105
DEPENDENCY GRAPH
112
SOFTWARE PIPELINING SECTION
106
INSTRUCTION SCHEDULE
113
LOOP UNROLLING SECTION
114
INSTRUCTION SCHEDULE
PREFETCH INSTRUCTION DELETING SECTION
107
115
PREFETCH INSTRUCTION RECOVERY SECTION
INSTRUCTION SCHEDULE
116
PREFETCH ADDRESS ADJUSTING SECTION
108
103
118
INSTRUCTION SCHEDULE
FLOW OF CONTROL
FLOW OF DATA

FIG. 1

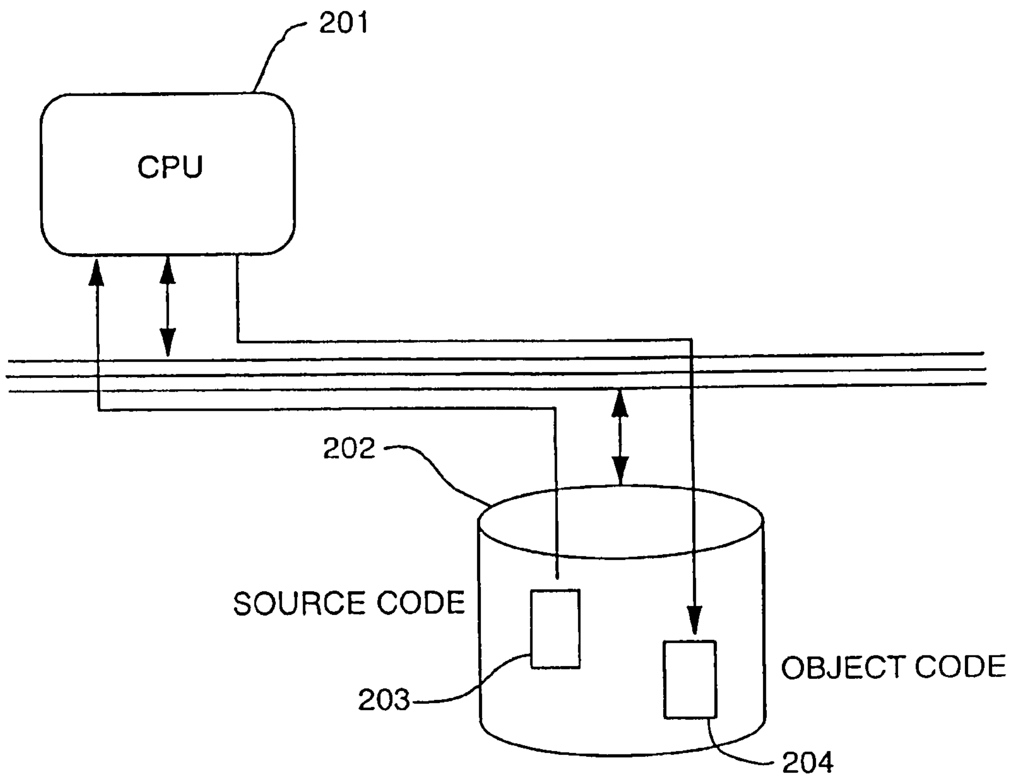

201
CPU
202
SOURCE CODE
203
OBJECT CODE
204

FIG. 2

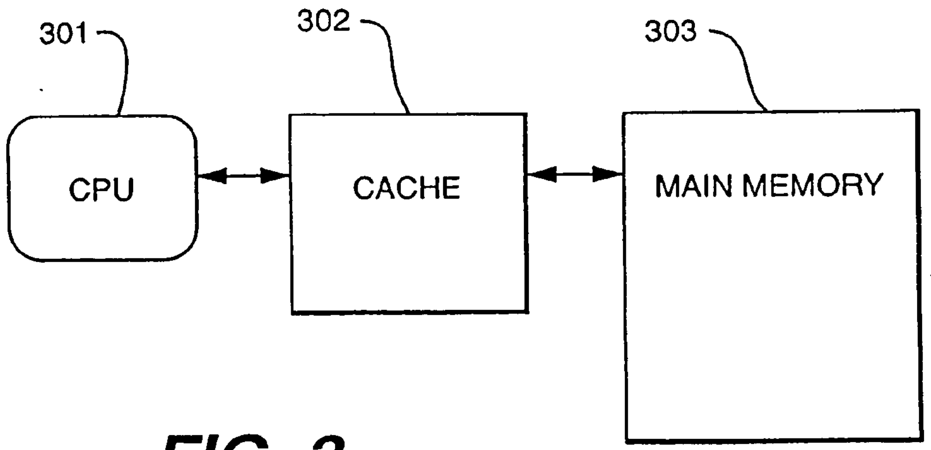

301
302
303
CPU
CACHE
MAIN MEMORY

FIG. 3

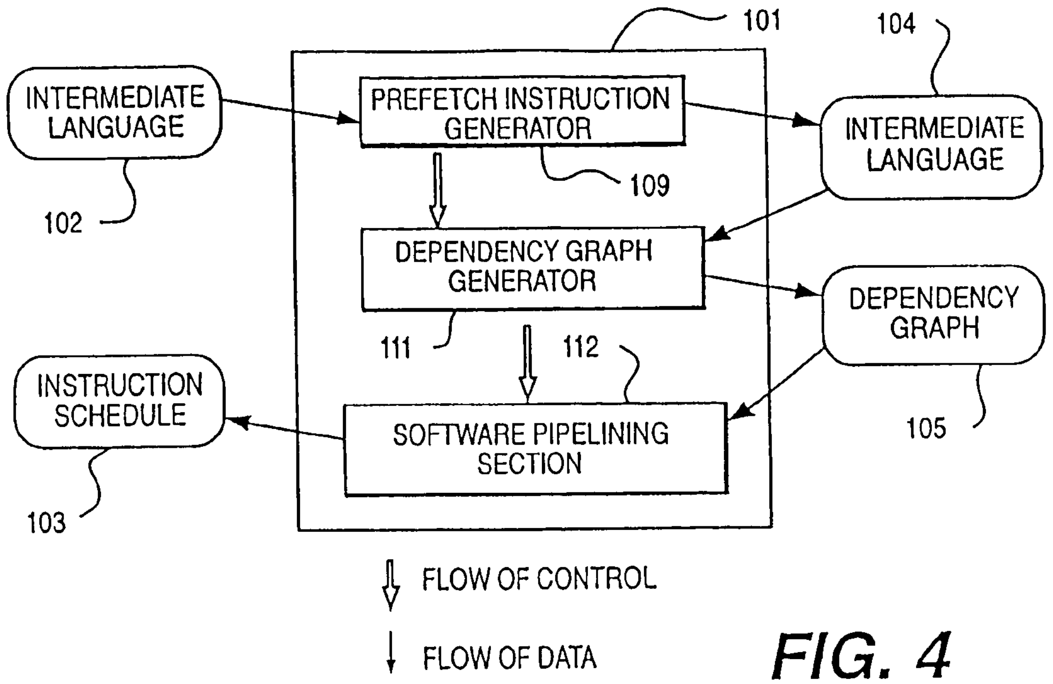

101
104
INTERMEDIATE LANGUAGE
PREFETCH INSTRUCTION GENERATOR
INTERMEDIATE LANGUAGE
109
102
DEPENDENCY GRAPH GENERATOR
DEPENDENCY GRAPH
111
112
INSTRUCTION SCHEDULE
SOFTWARE PIPELINING SECTION
105
103
FLOW OF CONTROL
FLOW OF DATA

FIG. 4

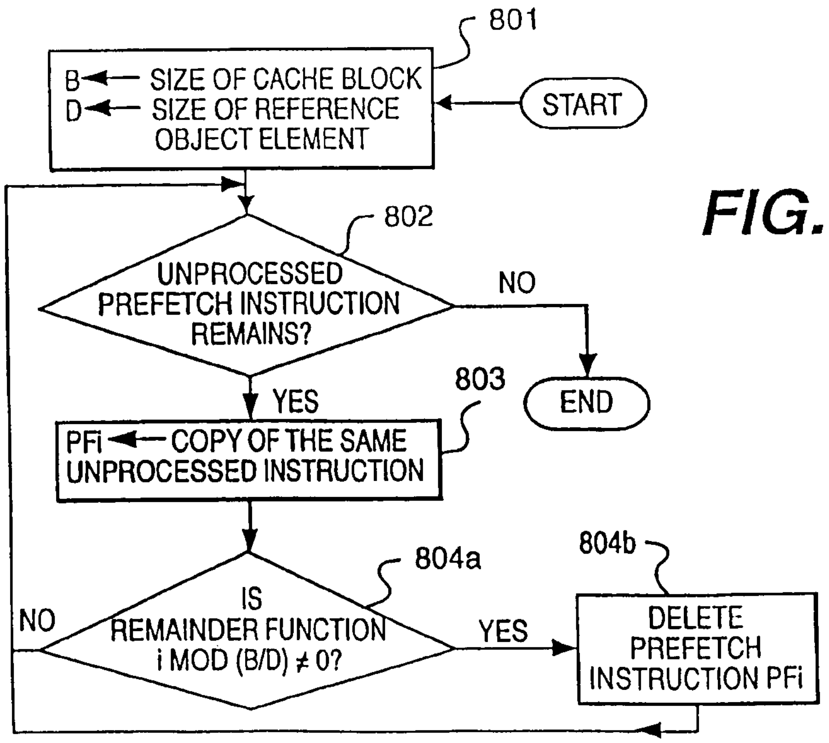

801
B← SIZE OF CACHE BLOCK
D← SIZE OF REFERENCE OBJECT ELEMENT
START
802
UNPROCESSED PREFETCH INSTRUCTION REMAINS?
NO
YES
803
END
PFi←COPY OF THE SAME UNPROCESSED INSTRUCTION
804a
804b
NO
IS REMAINDER FUNCTION i MOD (B/D) ≠ 0?
YES
DELETE PREFETCH INSTRUCTION PFi

FIG. 8

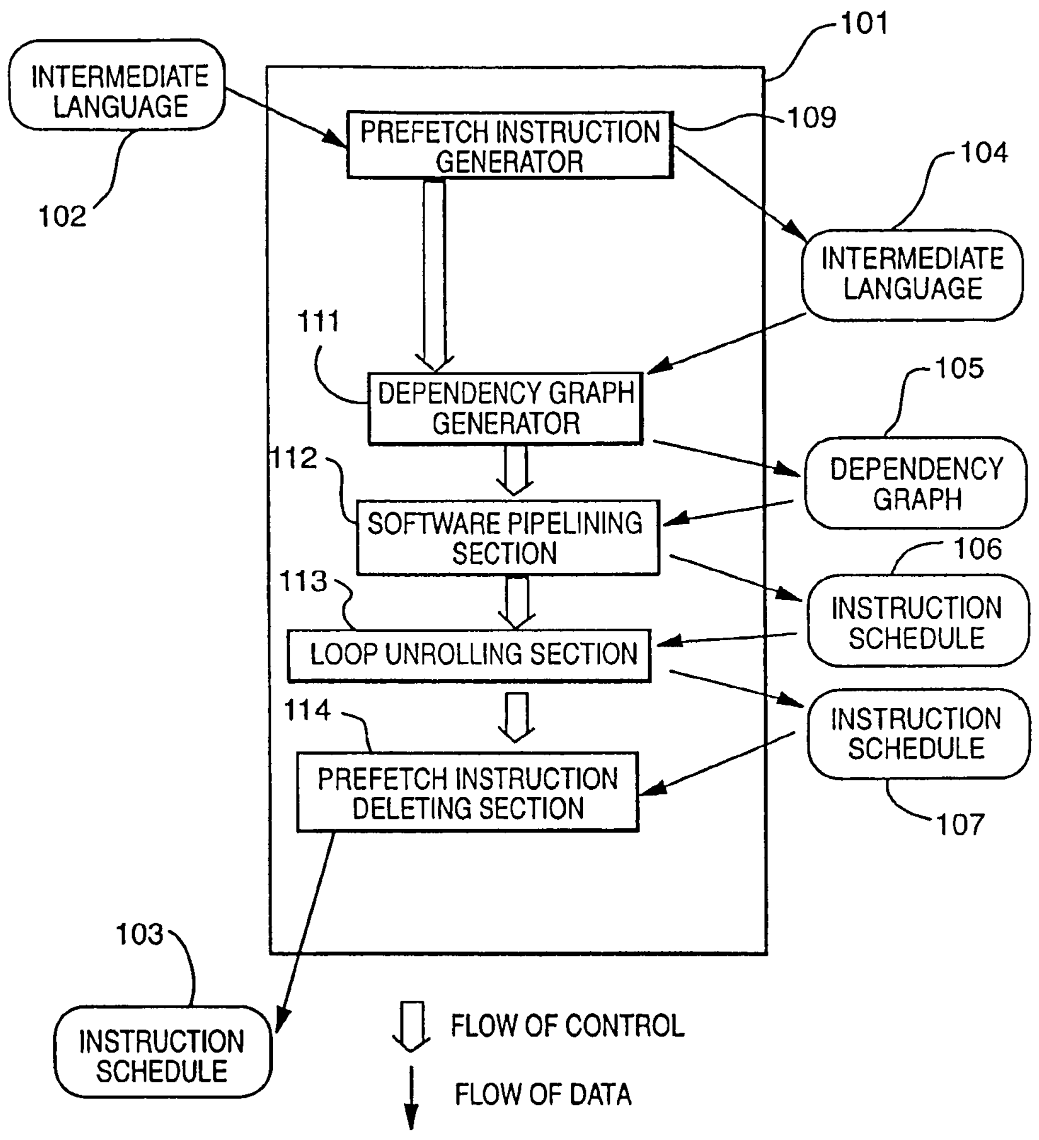

101
INTERMEDIATE LANGUAGE
102
PREFETCH INSTRUCTION GENERATOR
109
104
INTERMEDIATE LANGUAGE
111
DEPENDENCY GRAPH GENERATOR
105
DEPENDENCY GRAPH
112
SOFTWARE PIPELINING SECTION
106
INSTRUCTION SCHEDULE
113
LOOP UNROLLING SECTION
INSTRUCTION SCHEDULE
114
PREFETCH INSTRUCTION DELETING SECTION
107
103
INSTRUCTION SCHEDULE
FLOW OF CONTROL
FLOW OF DATA

FIG. 5

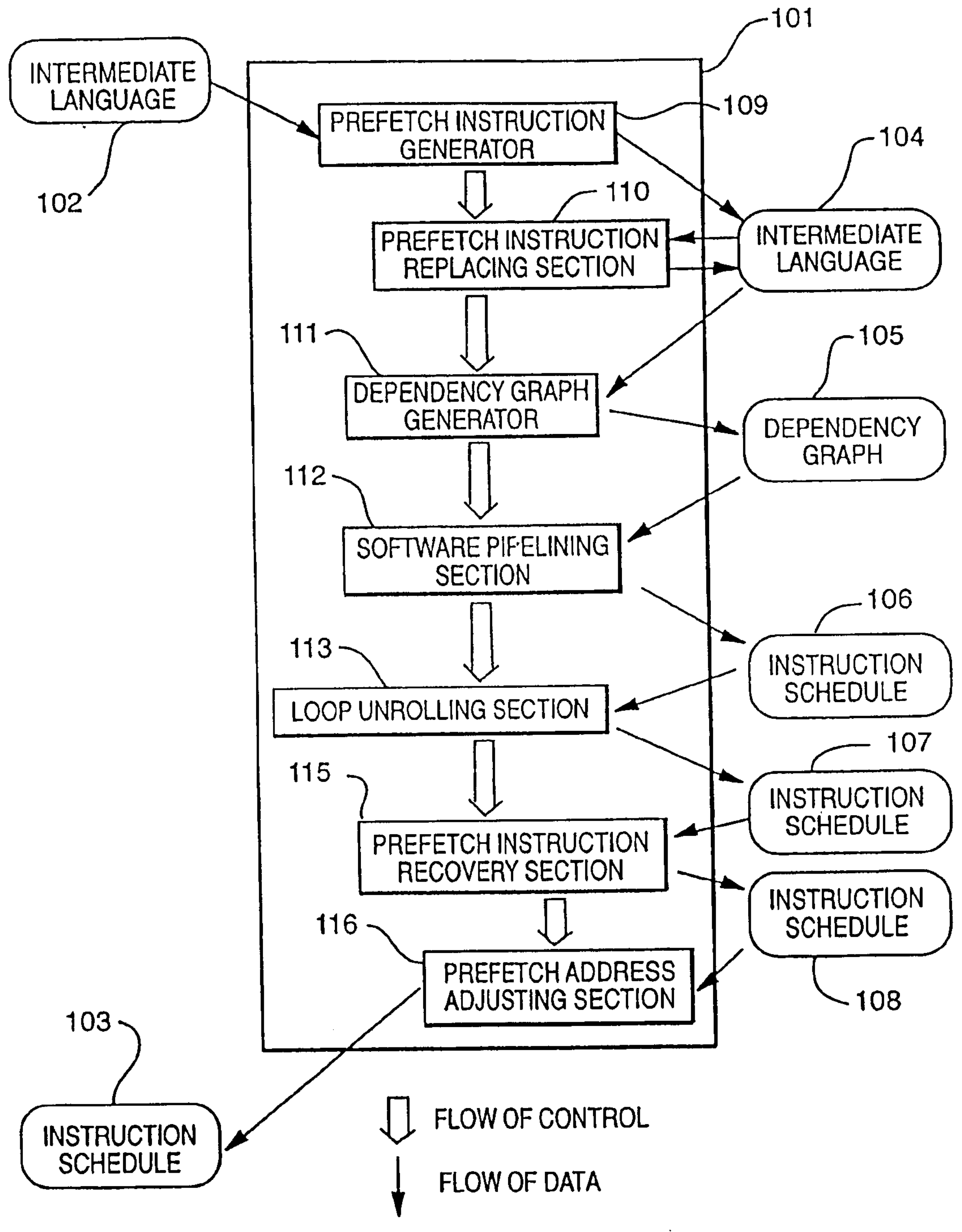

101
INTERMEDIATE LANGUAGE
102
PREFETCH INSTRUCTION GENERATOR
109
104
PREFETCH INSTRUCTION REPLACING SECTION
110
INTERMEDIATE LANGUAGE
111
DEPENDENCY GRAPH GENERATOR
105
DEPENDENCY GRAPH
112
SOFTWARE PIPELINING SECTION
106
INSTRUCTION SCHEDULE
113
LOOP UNROLLING SECTION
107
INSTRUCTION SCHEDULE
115
PREFETCH INSTRUCTION RECOVERY SECTION
INSTRUCTION SCHEDULE
116
PREFETCH ADDRESS ADJUSTING SECTION
108
103
INSTRUCTION SCHEDULE
FLOW OF CONTROL
FLOW OF DATA

FIG. 6

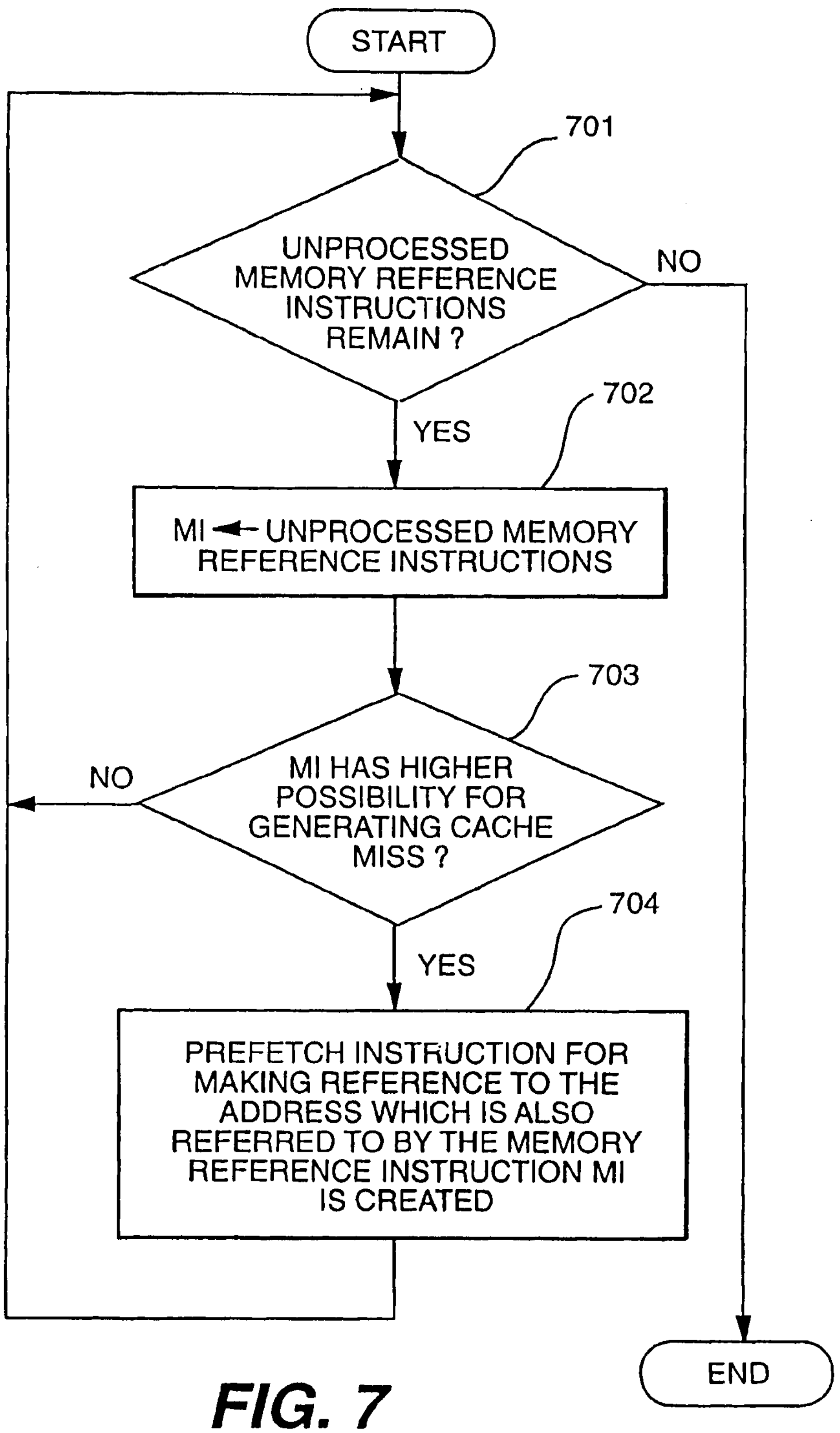

START
701
UNPROCESSED MEMORY REFERENCE INSTRUCTIONS REMAIN ?
NO
YES
702
MI ← UNPROCESSED MEMORY REFERENCE INSTRUCTIONS
703
MI HAS HIGHER POSSIBILITY FOR GENERATING CACHE MISS ?
NO
YES
704
PREFETCH INSTRUCTION FOR MAKING REFERENCE TO THE ADDRESS WHICH IS ALSO REFERRED TO BY THE MEMORY REFERENCE INSTRUCTION MI IS CREATED
END

FIG. 7

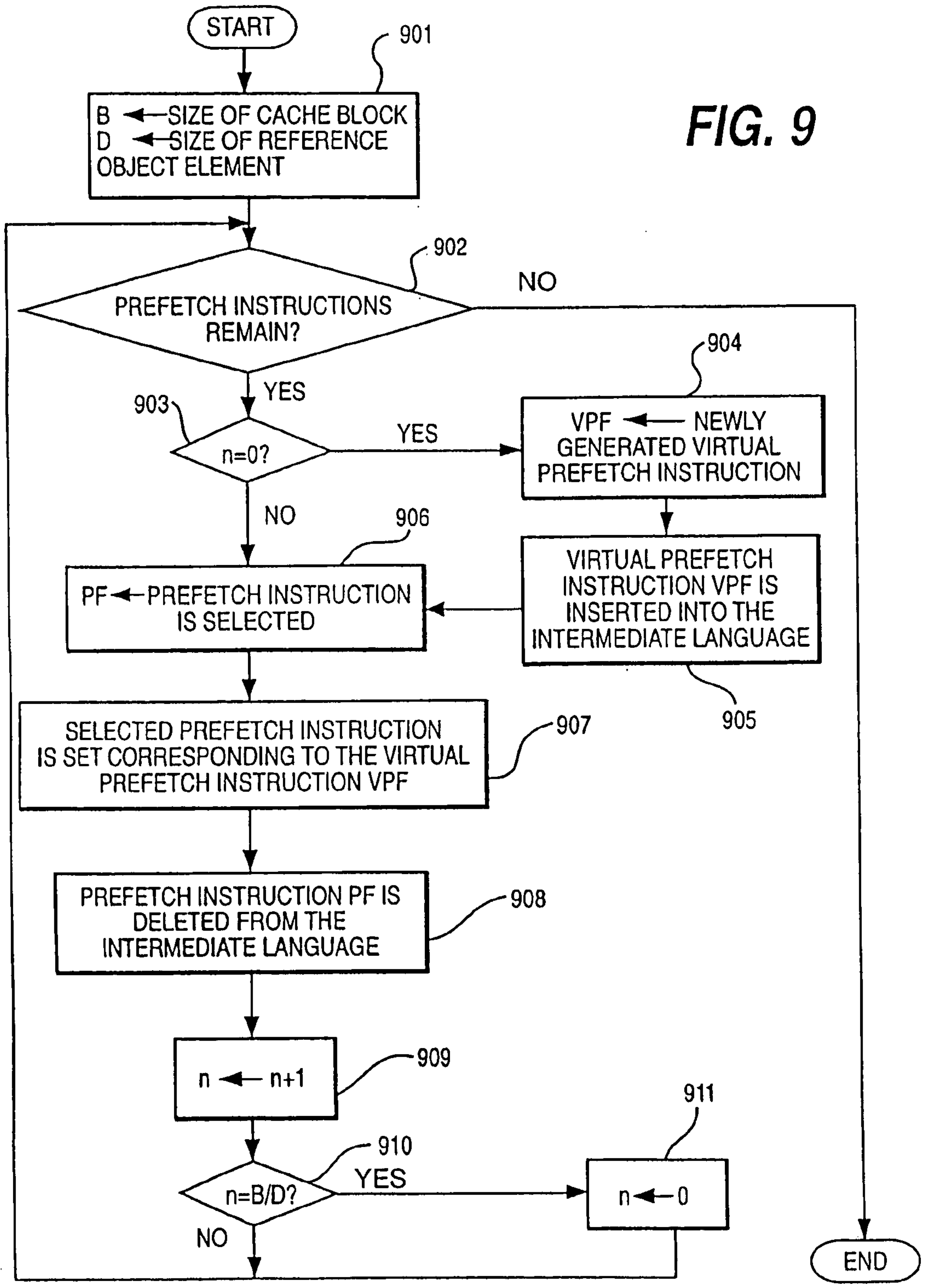

START
901
B ← SIZE OF CACHE BLOCK
D ← SIZE OF REFERENCE OBJECT ELEMENT
902
PREFETCH INSTRUCTIONS REMAIN?
NO
YES
903
n=0?
YES
904
VPF ← NEWLY GENERATED VIRTUAL PREFETCH INSTRUCTION
NO
906
PF ← PREFETCH INSTRUCTION IS SELECTED
VIRTUAL PREFETCH INSTRUCTION VPF IS INSERTED INTO THE INTERMEDIATE LANGUAGE
905
SELECTED PREFETCH INSTRUCTION IS SET CORRESPONDING TO THE VIRTUAL PREFETCH INSTRUCTION VPF
907
PREFETCH INSTRUCTION PF IS DELETED FROM THE INTERMEDIATE LANGUAGE
908
n ← n+1
909
911
910
n=B/D?
YES
n ← 0
NO
END

FIG. 9

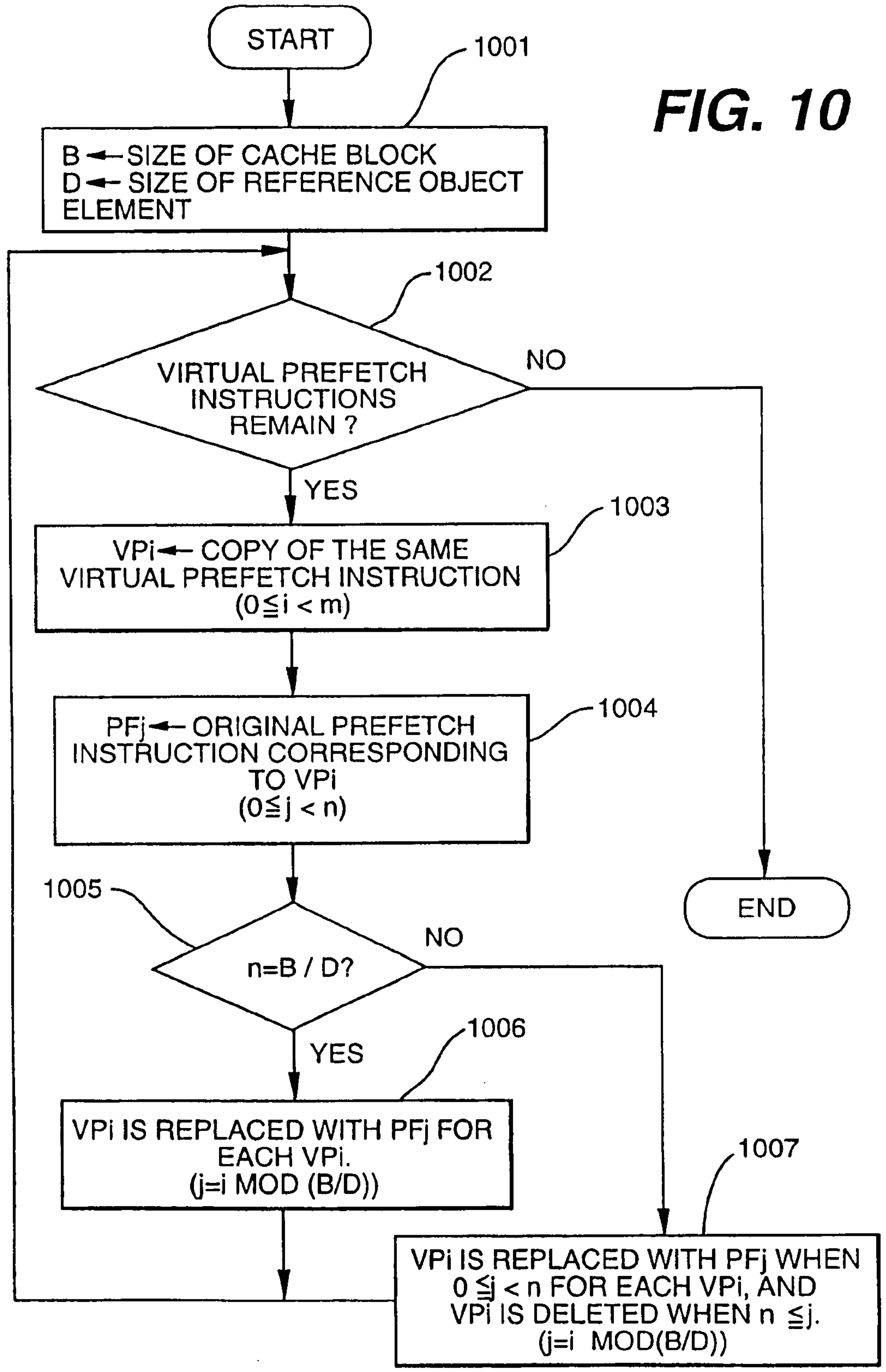

START
1001
B ← SIZE OF CACHE BLOCK
D ← SIZE OF REFERENCE OBJECT ELEMENT
1002
VIRTUAL PREFETCH INSTRUCTIONS REMAIN ?
NO
YES
1003
VPi ← COPY OF THE SAME VIRTUAL PREFETCH INSTRUCTION ($0 \leqq i < m$)
1004
PFj ← ORIGINAL PREFETCH INSTRUCTION CORRESPONDING TO VPi ($0 \leqq j < n$)
1005
n=B / D?
NO
YES
1006
VPi IS REPLACED WITH PFj FOR EACH VPi. (j=i MOD (B/D))
1007
VPi IS REPLACED WITH PFj WHEN $0 \leqq j < n$ FOR EACH VPi, AND VPi IS DELETED WHEN $n \leqq j$. (j=i MOD(B/D))
END

FIG. 10

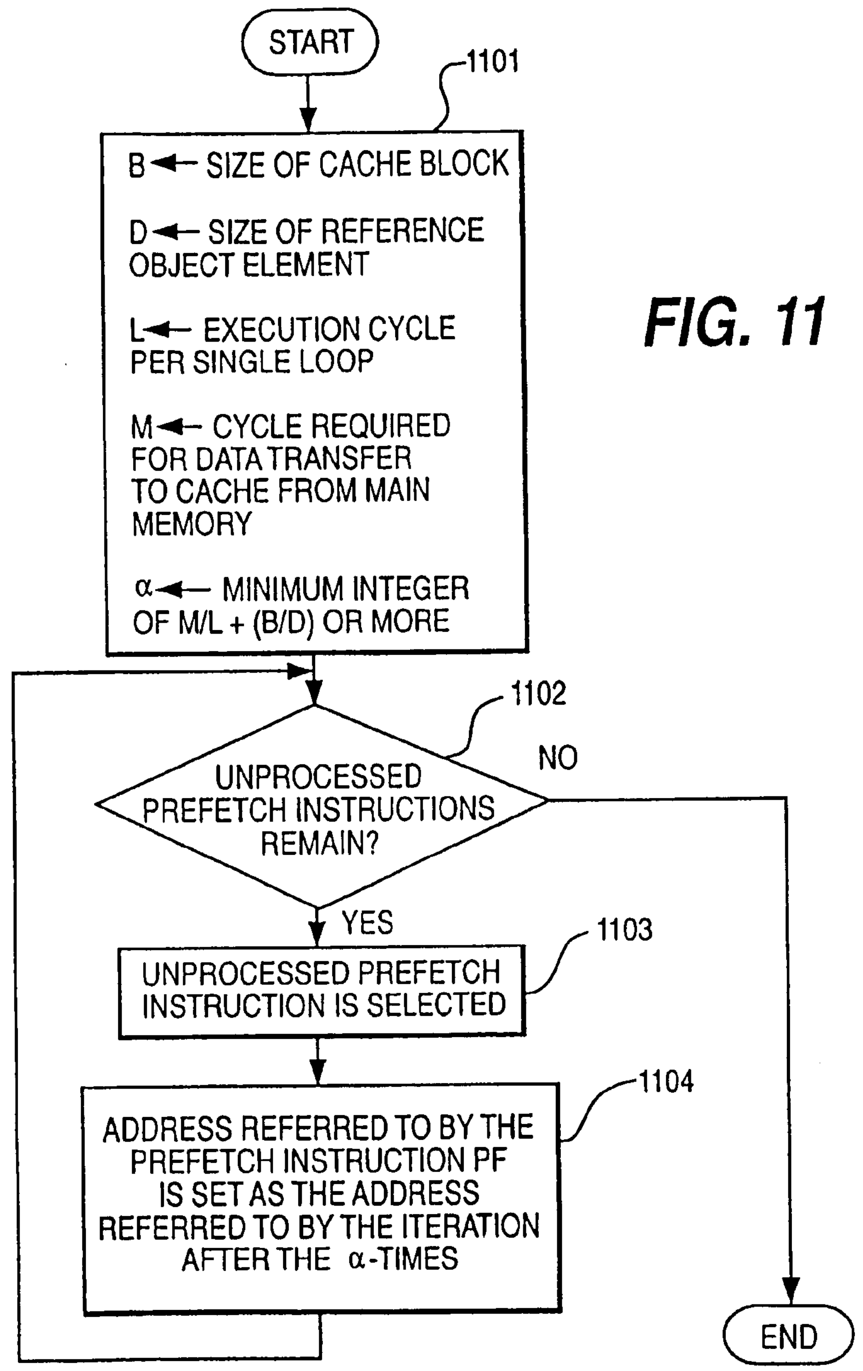

START
1101
B ← SIZE OF CACHE BLOCK
D ← SIZE OF REFERENCE OBJECT ELEMENT
L ← EXECUTION CYCLE PER SINGLE LOOP
M ← CYCLE REQUIRED FOR DATA TRANSFER TO CACHE FROM MAIN MEMORY
α ← MINIMUM INTEGER OF M/L + (B/D) OR MORE
1102
UNPROCESSED PREFETCH INSTRUCTIONS REMAIN?
NO
YES
1103
UNPROCESSED PREFETCH INSTRUCTION IS SELECTED
1104
ADDRESS REFERRED TO BY THE PREFETCH INSTRUCTION PF IS SET AS THE ADDRESS REFERRED TO BY THE ITERATION AFTER THE α-TIMES
END

FIG. 11

```
DO 10 I-0, N
   X(I) = A*X( I) + Y(I)
10 CONTINUE
```

FIG. 12

FIG. 13

```
t0 = X[i]         1301
t1 = A*t0
t2 = Y[i]         1302
t3 = t1 + t2
X[i] = t3         1303
```

FIG. 14

```
fetch X[i]        1401
fetch Y[i]        1402
t0 = X[i]
t1 = A*t0
t2 = Y[i]
t3 = t1 + t2
X[i] = t3
```

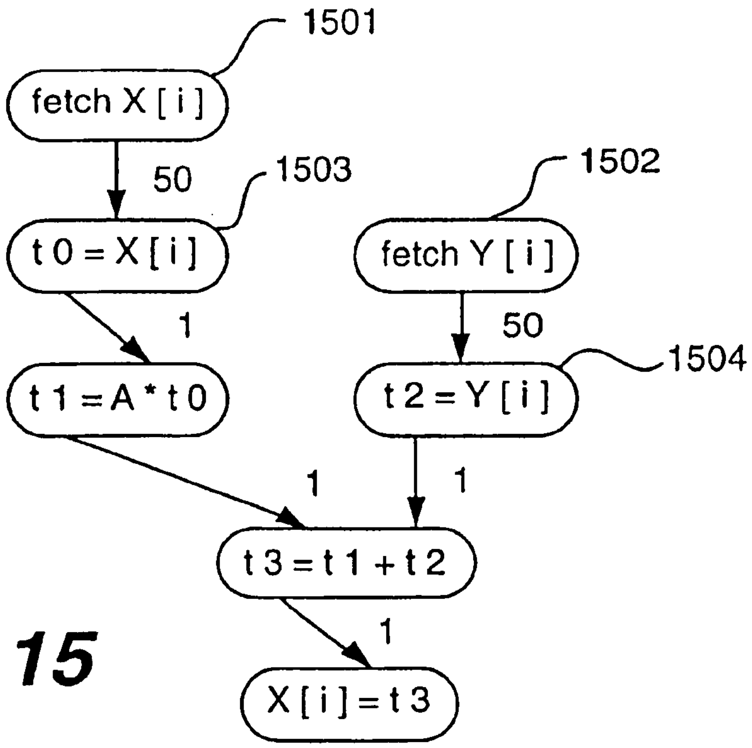


FIG. 15

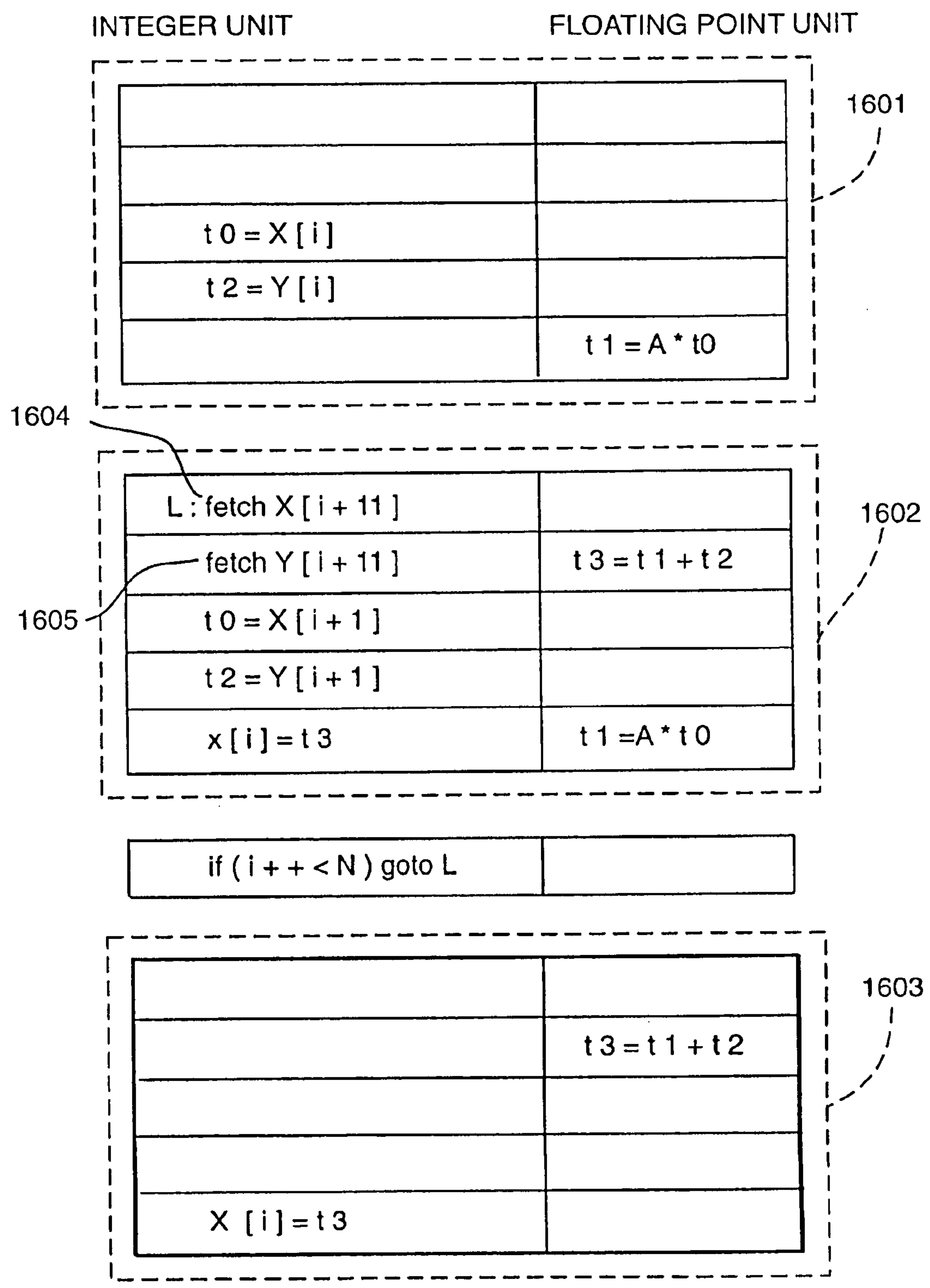

INTEGER UNIT
FLOATING POINT UNIT
t0 = X[i]
t2 = Y[i]
t1 = A * t0
1601
1604
L: fetch X[i + 11]
fetch Y[i + 11]
t3 = t1 + t2
1605
t0 = X[i + 1]
t2 = Y[i + 1]
x[i] = t3
t1 = A * t0
1602
if (i++ < N) goto L
t3 = t1 + t2
X [i] = t3
1603

FIG. 16

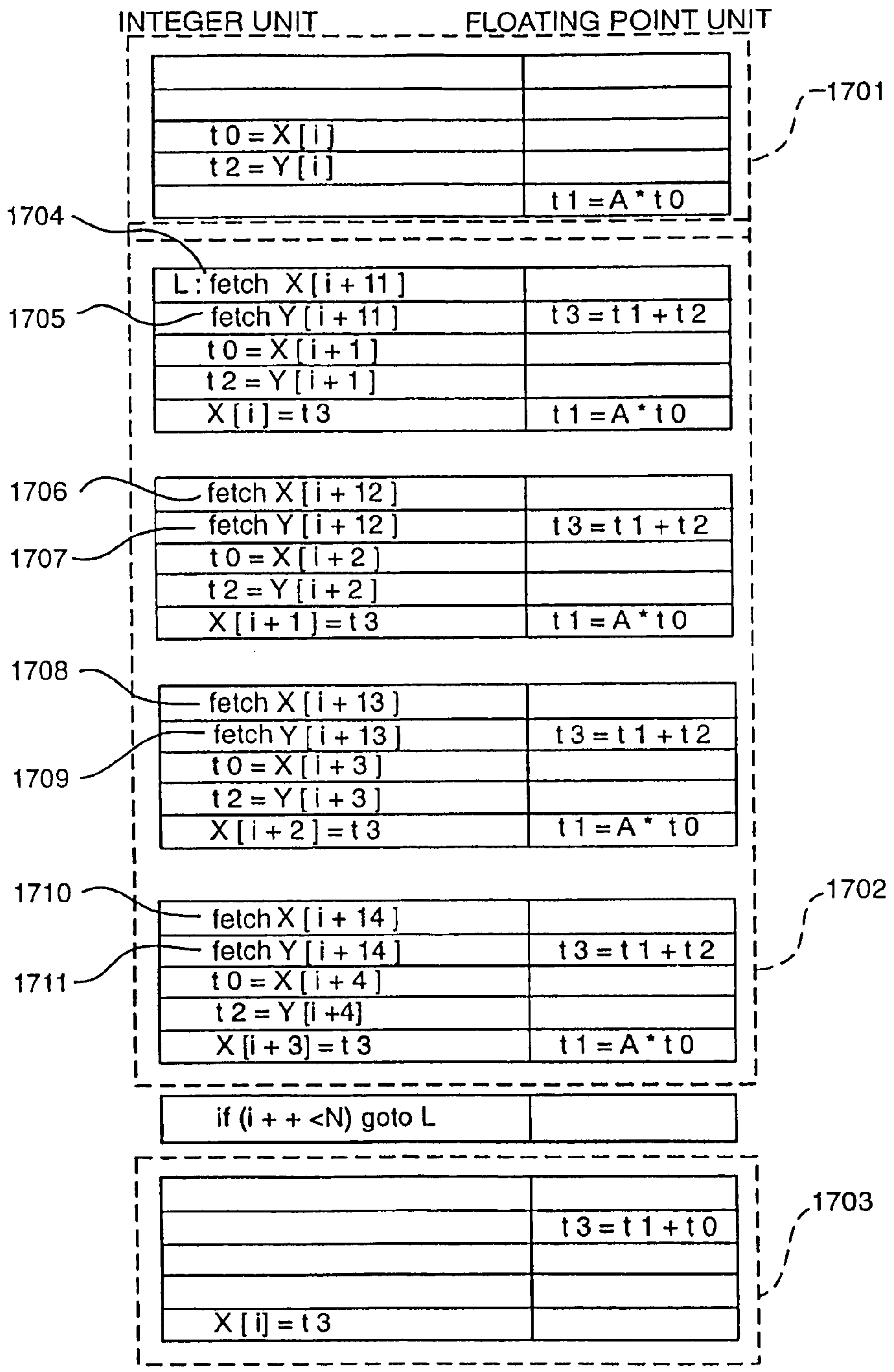

INTEGER UNIT
FLOATING POINT UNIT
1701
t0=X[i]
t2=Y[i]
t1=A*t0
1704
L: fetch X[i+11]
1705
fetch Y[i+11]
t3=t1+t2
t0=X[i+1]
t2=Y[i+1]
X[i]=t3
t1=A*t0
1706
fetch X[i+12]
fetch Y[i+12]
1707
t3=t1+t2
t0=X[i+2]
t2=Y[i+2]
X[i+1]=t3
t1=A*t0
1708
fetch X[i+13]
fetch Y[i+13]
1709
t3=t1+t2
t0=X[i+3]
t2=Y[i+3]
X[i+2]=t3
t1=A* t0
1710
fetch X[i+14]
fetch Y[i+14]
1711
t3=t1+t2
t0=X[i+4]
t2=Y[i+4]
X[i+3]=t3
t1=A*t0
1702
if (i++<N) goto L
1703
t3=t1+t0
X[i]=t3

FIG. 17

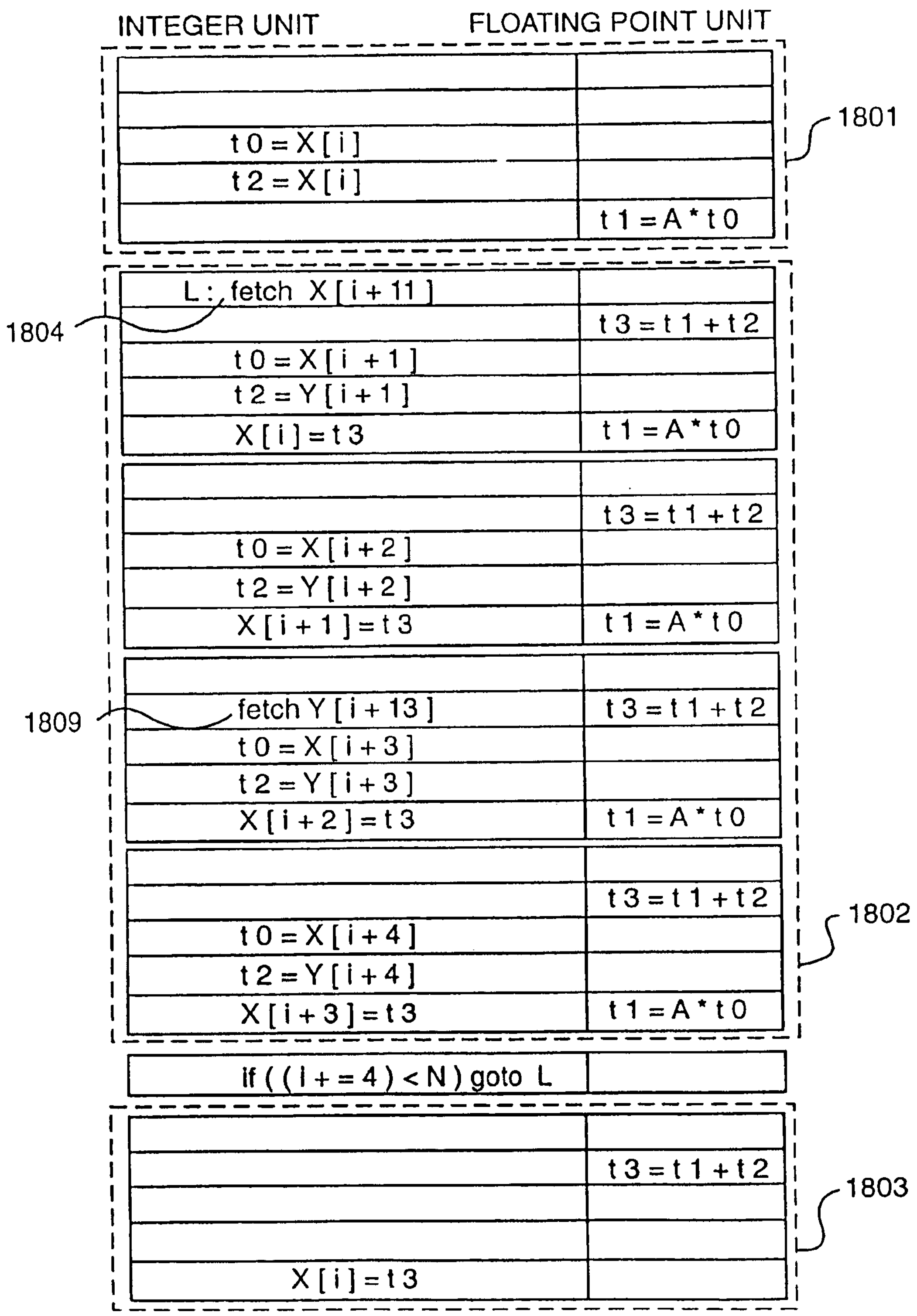

INTEGER UNIT
FLOATING POINT UNIT
1801
t0 = X[i]
t2 = X[i]
t1 = A*t0
L: fetch X[i+11]
1804
t3 = t1 + t2
t0 = X[i +1]
t2 = Y[i+1]
X[i] = t3
t1 = A*t0
t3 = t1 + t2
t0 = X[i+2]
t2 = Y[i+2]
X[i+1] = t3
t1 = A*t0
1809
fetch Y[i+13]
t3 = t1 + t2
t0 = X[i+3]
t2 = Y[i+3]
X[i+2] = t3
t1 = A*t0
t3 = t1 + t2
1802
t0 = X[i+4]
t2 = Y[i+4]
X[i+3] = t3
t1 = A*t0
if ((i += 4) < N) goto L
t3 = t1 + t2
1803
X[i] = t3

FIG. 18

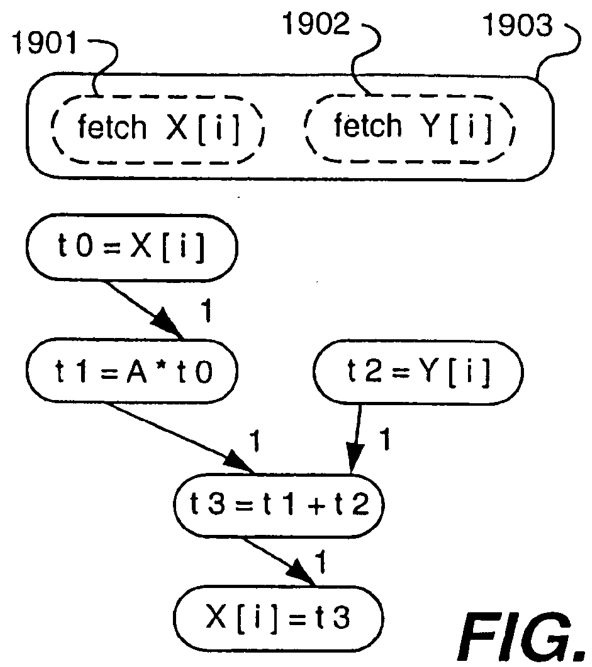

1901
1902
1903
fetch X[i]
fetch Y[i]
t0=X[i]
1
t1=A*t0
t2=Y[i]
1
1
t3=t1+t2
1
X[i]=t3

FIG. 19

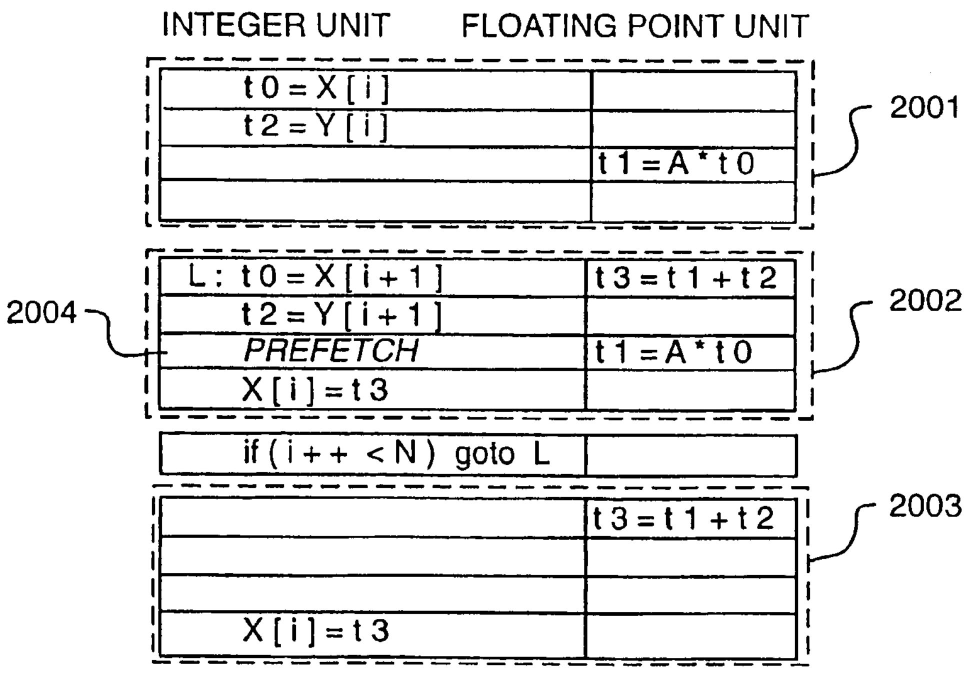

INTEGER UNIT
FLOATING POINT UNIT
t0=X[i]
t2=Y[i]
t1=A*t0
2001
L: t0=X[i+1]
t3=t1+t2
t2=Y[i+1]
PREFETCH
t1=A*t0
X[i]=t3
2004
2002
if(i++ <N) goto L
t3=t1+t2
X[i]=t3
2003

FIG. 20

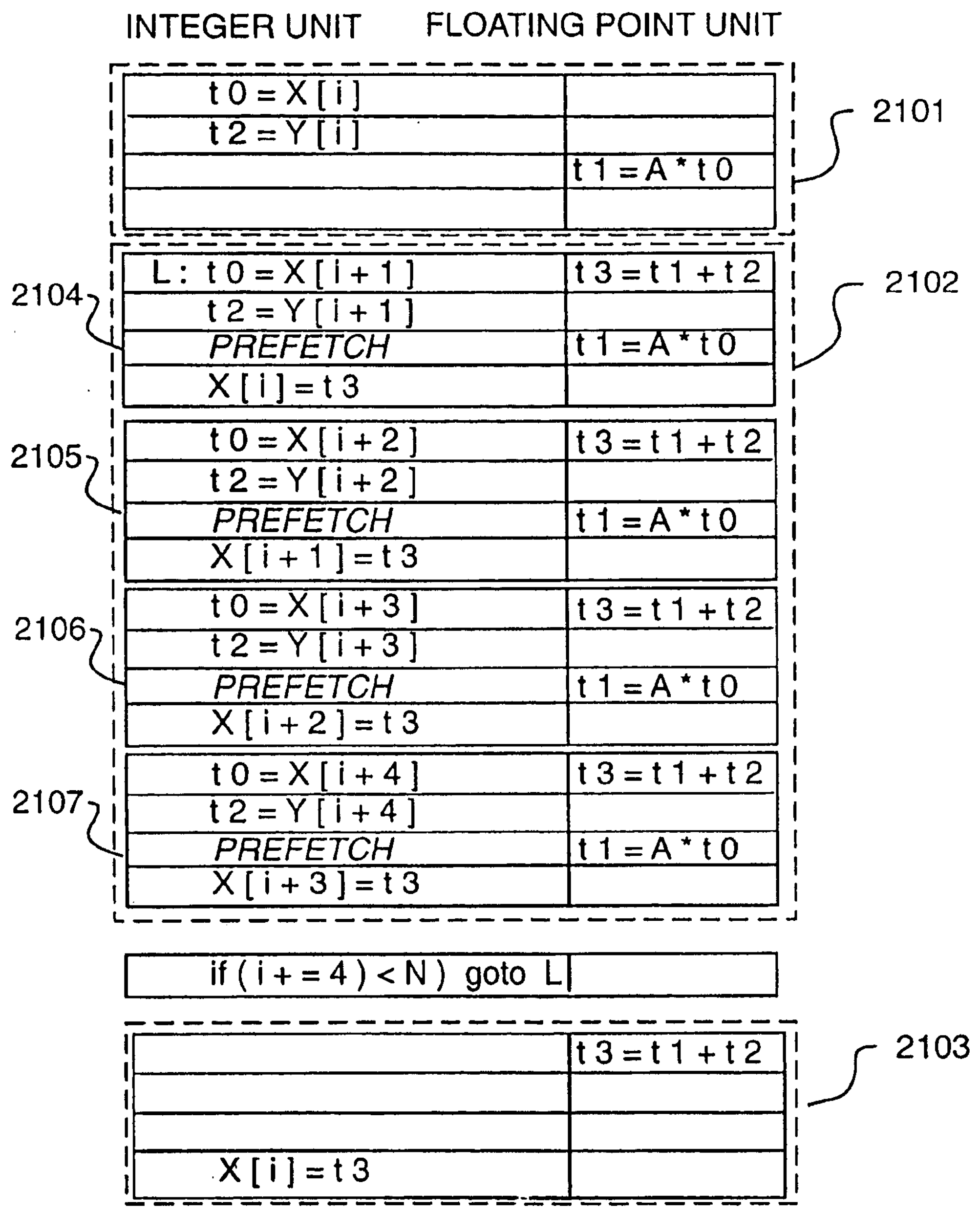
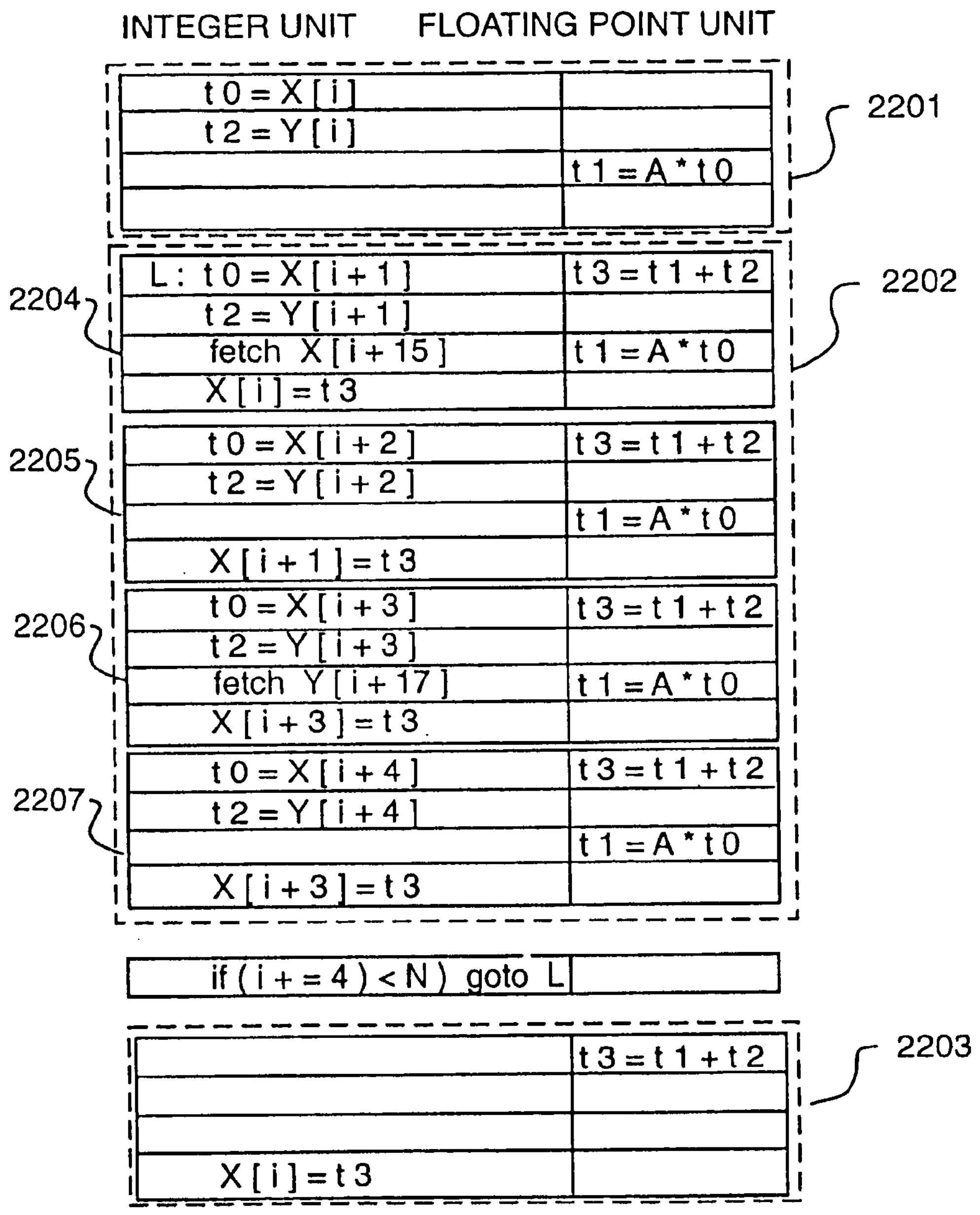

INTEGER UNIT
FLOATING POINT UNIT
t0=X[i]
t2=Y[i]
t1=A*t0
2101
L: t0=X[i+1]
t3=t1+t2
t2=Y[i+1]
PREFETCH
t1=A*t0
X[i]=t3
2102
2104
t0=X[i+2]
t3=t1+t2
t2=Y[i+2]
PREFETCH
t1=A*t0
X[i+1]=t3
2105
t0=X[i+3]
t3=t1+t2
t2=Y[i+3]
PREFETCH
t1=A*t0
X[i+2]=t3
2106
t0=X[i+4]
t3=t1+t2
t2=Y[i+4]
PREFETCH
t1=A*t0
X[i+3]=t3
2107
if (i+=4) < N) goto L
t3=t1+t2
X[i]=t3
2103

FIG. 21



INTEGER UNIT
FLOATING POINT UNIT
t0 = X[i]
t2 = Y[i]
t1 = A * t0
2201
L: t0 = X[i+1]
t3 = t1 + t2
t2 = Y[i+1]
fetch X[i+15]
t1 = A * t0
X[i] = t3
2202
2204
t0 = X[i+2]
t3 = t1 + t2
t2 = Y[i+2]
t1 = A * t0
X[i+1] = t3
2205
t0 = X[i+3]
t3 = t1 + t2
t2 = Y[i+3]
fetch Y[i+17]
t1 = A * t0
X[i+3] = t3
2206
t0 = X[i+4]
t3 = t1 + t2
t2 = Y[i+4]
t1 = A * t0
X[i+3] = t3
2207
if (i += 4) < N) goto L
t3 = t1 + t2
X[i] = t3
2203

FIG. 22

METHOD FOR COMPILING LOOPS CONTAINING PREFETCH INSTRUCTIONS THAT REPLACES ONE OR MORE ACTUAL PREFETCHES WITH ONE VIRTUAL PREFETCH PRIOR TO LOOP SCHEDULING AND UNROLLING

FIELD OF THE INVENTION

The present invention relates to a data prefetch method and more specifically to a compile method which shortens the execution time of a program by prefetching data through scheduling of the prefetch instruction for a loop.

BACKGROUND OF THE INVENTION

The executing time of a program depends significantly on the waiting time generated by the dependent relationship between instructions and the waiting time generated by memory references.

The waiting time generated by the dependent relationship between instructions within a loop can be considerably reduced by using a software pipelining scheduling method. Software pipelining as described, for example, in "Software Pipelining in PA-RISC Compiler" by S. Ramakrishnan, Hewlett-Packard Journal, pp. 39–45, 1992, reduces the waiting time generated by the dependent relationship between instructions and enhances the degree of parallelism in execution of instructions by overlapped execution of different iterations of the loop. The loop to which the software pipelining is applied is characterized by executing the code for initialization called a prologue before starting execution of the loop, executing the loop body by repeating code called a kernel, terminating the process by executing code called an epilogue when execution of the loop is completed, and starting execution of the subsequent iteration without waiting for the completion of the preceding iteration.

It is rather difficult, in comparison with the waiting time generated by the dependent relationship between instructions, to reduce the waiting time associated with the memory references only with a software method. Therefore, in many computer systems, a high speed and small capacity memory called a cache memory is provided between the main memory and a processor to reduce the waiting time generated by a memory reference and thereby a high speed reference can be made on the cache memory to the data referred to recently. However, even when a cache memory is used, the waiting time is inevitably generated if a cache miss occurs while there is no recycle use of data.

Therefore, as described, for example, in "Design and Evaluation of a Compiler Algorithm for Prefetching" by T. C. Mowry, et al., Proceedings of the 5th International Conference on Architectural Support for Programming Language and Operating Systems, pp. 62–73, 1992 for example, an attempt is made to reduce the waiting time generated by the memory references by utilizing an instruction for prefetching data from the main memory to the cache memory.

SUMMARY OF THE INVENTION

In the prior art, described above, software pipelining has been applied, as a method of scheduling a prefetch instruction, in such a manner that the prefetch instruction is issued before the iterations preceding a value that is a minimum integer times larger than the value obtained by dividing the delay time of the prefetch instruction with the

shortest path length of the loop body. However, the details for realizing such an application in such a manner are not yet described.

It is therefore an object of the present invention to provide an effective instruction scheduling method which can reduce the waiting time generated by a memory reference and the waiting time generated by the dependent relationship between instructions (inter-instruction dependency) while a program is executed in the loop including the prefetch instruction.

In view of achieving the objects of the method of the present invention, the scheduling of the prefetch instruction for the loop in a program is executed in accordance with one of three types of methods at the time of compiling the program.

The value of the data is not altered during the prefetch of data to the cache. Therefore, depending on the ordinary relationship between the definition and the use of the data, a dependency relationship does not exist between the prefetch of data to the cache and reference to the memory with a load instruction or store instruction. However, it is convenient and advantageous, because the existing scheduling system can be applied directly, for hiding the waiting time due to the reference to memory, when a tacit dependent relationship is assumed to exist between the prefetch instruction and the instruction for making reference to the memory due to the limitation that the memory reference instruction must be issued after completion of the data transfer to the cache by the prefetch instruction. Therefore, in method 1, the scheduling is performed by providing the dependent relationship between the prefetch instruction and the memory reference instruction.

Method 1

(1) The prefetch instruction is respectively issued for memory reference instructions which are assumed to generate a cache miss.

(2) A dependency graph having edges between the prefetch instructions generated in item (1) explained above and the corresponding memory reference instructions is generated. In this case, a delay between the prefetch instruction and the memory reference instruction is set to a value larger than the number of cycles required for data transfer to the cache by the prefetch instruction so that the memory reference instruction is issued after the number of cycles required for data transfer to the cache by the prefetch instruction.

(3) An instruction schedule is obtained by applying the software pipelining to the dependency graph generated in item (2) explained above. As explained above, software pipelining is a method that reduces the waiting time generated by the inter-instruction dependency between instructions by the overlapping execution of different iterations of the loop so that a sufficient interval can be provided between the prefetch instruction and the corresponding memory reference instruction. Thus, the scheduling is obtained by applying the software pipelining to the dependency graph generated in the item (2).

While data transfer to the cache from the main memory is generally carried out in units of 32 bytes or 128 bytes, etc., the reference to the array in the loop is often carried out in smaller units, such as 4 bytes or 8 bytes, etc. Therefore, when a memory reference is continuously carried out for the array, etc. in the loop, the data for reference can often be moved to the cache from the main memory with a plurality of repetitive executions of a prefetch instruction. That is, if it is possible to move, to the cache from the main memory, the data for reference with N times of repetitive execution of a prefetch instruction, it is enough that the prefetch instruction is generated once for every N times of execution.

For the schedule generated in method 1, since the prefetch instruction is generated once for every iteration, many redundant prefetch instructions are generated. Therefore, the prefetch instructions are scheduled by unrolling the loop so that the redundant prefetch instructions are not generated frequently.

In a method (2), the kernel section of the loop including the software pipelined prefetch instructions generated by the processings up to the item (3) from item (1) is unrolled unrolled to avoid issuing useless prefetch instructions by eliminating the redundant prefetch instructions.

Method 2

(4) Since it is sufficient to issue a prefetch instruction once for every iteration of N times when the number of data prefetched by the one prefetch instruction, the kernel section of the software pipelined schedule (item (3) explained above) is unrolled until the number of unrollings becomes a multiple of N.

(5) In the unrolled code of the item (4) explained above, the kernel section is unrolled for the number of times which is equal to a multiple of N and the iteration is executed for the number of times of a multiple of N of the loop with only one iteration of the kernel section unrolled. Therefore, issuing useless prefetch instructions can be eliminated by deleting the redundant prefetch instructions from the unrolled code so that the prefetch instruction is issued only once for an iteration of N times.

In method 2, since the redundant prefetch instructions are removed after the software pipelining is applied to the prefetch instruction, the interval between the instructions becomes shorter than that expected when the software pipelining is applied by deleting the prefetch instruction and thereby the waiting time caused by the inter-instruction dependency may become visible.

In a method 3, a plurality of prefetch instructions are replaced with one virtual prefetch instruction to generate a dependency graph including such a virtual prefetch instruction considering the unrolling of the kernel section after application of software pipeling to the loop. However, unlike methods 1 and 2, the dependent relationship may not be provided in method 3 between the virtual prefetch instruction and the corresponding memory reference instruction.

Next, software pipelining is applied to a dependency graph to obtain a software pipelined schedule and the loop is unrolled, as required, so that the number of times of unrolling of the kernel section becomes equal to a multiple of the number of data which can be prefetched by one prefetch instruction. The unrolled virtual prefetch instruction is replaced with the initial prefetch instruction to adjust an address referred to by the prefetch instruction so that the prefetch instructions are issued in an iteration sufficiently preceding the corresponding memory reference instruction.

Thereby, the dependent relationship between instructions generated by deleting the instruction in method 2 can be reduced.

Method 3

In accordance with method 3 of the present invention, the following steps are executed.

(1) The prefetch instructions are generated respectively for the memory reference instructions which are assumed to generate a cache miss.

(2) The prefetch instructions generated in item (1) explained above are grouped into a plurality of groups and are then replaced with virtual prefetch instructions.

(3) A dependency graph composed of an instruction of an original loop body and virtual prefetch instructions generated in item (2) above is generated and the software pipelining is applied thereto. In the case of generating the dependency graph, it is no longer necessary to think about the dependent relationship between the virtual prefetch instruction and the corresponding memory reference instruction.

(4) The loop is unrolled as required so that the number of times of unrolling of the kernel section formed in the item (3) explained above becomes equal to a multiple of the number of times of iteration for prefetching data with one prefetch instruction. In the schedule after the unrolling, the virtual prefetch instruction indicates instruction slots, in any one of which the original prefetch instruction is inserted.

(5) The virtual prefetch instruction scheduled in the unrolled code of item (4) above is replaced with the original prefetch instruction. This replacement is necessary to issue the same prefetch instruction for every multiple of the number of iterations for prefetching the data with one prefetch instruction. Thereby, issuing redundant prefetch instructions can be controlled.

(6) An address used for reference by the prefetch instruction replaced in the item (5) explained above is used as the address of the data referred to subsequently in the completion of the data transfer by means of the prefetch instruction.

According to the method of the present invention, if reference to memory is not continuously executed, an interval between the prefetch instruction and a memory reference instruction can be maintained sufficiently long by method 1 in view of applying the software pipelining. Moreover, when the memory reference is executed continuously, issuing redundant prefetch instructions can be controlled to effectively perform the scheduling by removing the instruction after application of the software pipelining by means of method 2 or by applying the software pipelining through replacement of a plurality of prefetch instructions with the virtual prefetch instruction by means of method 3 and then recovering the virtual prefetch instruction into the original prefetch instruction. Thereby, the object of the present invention can be achieved.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a diagram of an instruction scheduler for scheduling the prefetch instructions.

FIG. 2 is an example of a computer system in which the present invention is employed.

FIG. 3 is an example of a computer in which the present invention is employed.

FIG. 4 is a diagram of an instruction scheduler that executes scheduling according to method 1.

FIG. 5 is a diagram of an instruction scheduler that executes scheduling according to method 2.

FIG. 6 is a diagram of an instruction scheduler that executes scheduling according to method 3.

FIG. 7 is a flowchart of a prefetch instruction generator.

FIG. 8 is a flowchart of a prefetch instruction deleting section.

FIG. 9 is a flowchart of a prefetch instruction replacing section.

FIG. 10 is a flowchart of a prefetch instruction recovery section.

FIG. 11 is a flowchart of a prefetch address adjusting section.

FIG. 12 is an example of a FORTRAN source program.

FIG. 13 is an example of the intermediate language generated in compiling the source program of FIG. 12.

FIG. **14** is an example of the intermediate language of FIG. **13** which includes the prefetch instructions.

FIG. **15** is an example of a dependency graph for the intermediate language of FIG. **14** which includes the prefetch instructions according to method 1.

FIG. **16** is an example of the software-pipelined schedule obtained by applying software pipelining to the dependency graph of FIG. **15** according to method 1.

FIG. **17** is an example of the unrolled schedule obtained by method 2.

FIG. **18** is an example of the schedule of FIG. **17** having the redundant prefetch instructions deleted according to method 2.

FIG. 19 is an example of a dependency graph generated according to method 3.

FIG. **20** is an example of the software-pipelined schedule obtained by applying software pipelining to the dependency graph of FIG. **19** according to method 3.

FIG. **21** is an example of the unrolled schedule according to method 3.

FIG. 22 is an example of the schedule obtained following replacement of the prefetch instructions in method 3.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A preferred embodiment of the present invention will be explained with reference to the accompanying drawings.

FIG. **2** illustrates an example of a computer to which the method of the present invention is applied. In this example, a compiler, which in a preferred embodiment is constituted by a combination of software stored on a storage medium **202**, such as a hard disk or other storage device, and hardware, such as a computer (CPU) **201** that executes the software to perform the function of the compiler, operates on the CPU **201** and reads the source code **203** from an external memory **202**, converts it into the object code **204** and then stores the object code into the external memory **202**.

FIG. 3 illustrates an example of a computer in which the data prefetch method of the present invention is employed. In the case of executing an ordinary memory reference instruction with the CPU **301**, it is first checked whether the reference object data is in the cache **302**. When such data exists in the cache **302**, reference is made to such data. If there is no such data as the reference object, reference is made to the relevant data in the main memory **303** and a cache block to which the relevant data belongs is placed in the cache **302**. Reference to the cache is made at a high speed in comparison with the reference to the main memory and when the data as the reference object is found to be in the cache, the waiting time generated by the memory reference is reduced.

The prefetch instruction is used for moving the cache block that the data of the reference object belongs to into the cache **302** from the main memory **303** simultaneously with the execution of the other instructions. Since the other instructions can be executed during the transfer of the data to the cache **302** from the main memory **303** by issuing the prefetch instruction beforehand by a number of cycles sufficient for movement of the cache block to the cache **302** from the main memory **303**, the waiting time for making reference to the relevant data can be eliminated.

FIG. 1 illustrates a diagram providing an example of the present invention. In FIG. **1**, a scheduling processor **101** inputs an intermediate language **102** for a loop body and outputs an instruction schedule **103** including the prefetch

instructions and having a reduced amount of delay caused by inter-instruction dependency and a reduced amount of waiting time resulting from memory reference. Processings **117** and **118** (software program processings) are characteristic processings of the present invention. In the processing **117**, the generation of prefetch instructions and the preprocessing of the scheduling are executed, while in the processing **118**, the removal of redundant prefetch instructions and postprocessing, such as adjustment of prefetch addresses, are performed.

First, an embodiment for scheduling the loop with method 1 will be described. FIG. 4 is a diagram of an instruction scheduler for scheduling the prefetch instruction with method 1. In method 1, the prefetch instruction generator **109** inputs an intermediate language **102** to provide the intermediate language **104** having the added prefetch instructions by generating the prefetch instructions for the memory reference instructions which are assumed to have a high possibility for generating a cache miss among those included in the intermediate language **102** included in the loop body.

Here, the possibility for generating a cache miss of a certain memory reference instruction can be estimated according to the known prior art described in "Design and Evaluation of a Compiler Algorithm for Prefetching" by T. C. Mowry et al., Proceedings of the 5th International Conference on Architectural Support for Programming Language and Operating Systems, pp. 62–73, 1992, for example, along with a trace of the program execution. The addresses prefetched by the prefetch instructions generated are assumed to be those of the corresponding memory reference instructions.

Namely, if a load instruction, LOAD X[i], in the loop is assumed to easily generate a cache miss, an instruction for prefetching the same element, FETCH X[i], is generated and it is then added to the intermediate language.

Next, the dependent graph generator **111** inputs the intermediate language **104** including the prefetch instruction to generate a dependency graph **105**. In this case, an edge indicating that a delay required between the prefetch instruction and the corresponding memory reference instruction is longer than the time required for transferring the cache block to the cache from the main memory is provided between the prefetch instruction and the corresponding memory reference instruction. Next, the software pipelining is applied to the dependency graph **105** in the software pipelining section **112** to obtain the software pipelined instruction schedule **103**.

As explained above, since it is guaranteed that an interval between the prefetch instruction and the corresponding memory reference instruction is set to the time required for transfer of a cache block to the cache from the main memory at the time of application of the software pipelining by generating the dependency graph having an edge between the prefetch instruction and the corresponding memory reference instruction to indicate that the necessary delay is longer than the time required for transfer of the cache block to the cache from the main memory, the prefetch instruction can be scheduled to hide the latency due to the memory reference.

The prefetch instruction generator **109** explained above will be further explained with reference to an operation flowchart shown in FIG. 7. First, in step **701**, whether any memory reference instructions to be processed remain or not is judged. When such instruction is left, control skips to step **702**. When there is no such instruction, processing is completed. In step **702**, the memory reference instruction to be processed is selected and is then stored in a variable MI. In step **703**, whether the memory reference instruction stored in MI has a high possibility for generating a cache miss or not is judged. When such possibility for generating a cache miss is high, the control skips to step **704**. When such possibility is low, control skips to step **701** to process the next memory reference instruction. In step **704**, the prefetch instruction for making reference to the address that is the same as that of the memory reference instruction is stored in MI is generated.

Next, an embodiment for scheduling the loop by method 2 will be explained. In method 2, the following processings are also executed in addition to the processings of method 1. First, the kernel section of software pipelined instruction schedule **106** obtained by the processing of method **1** is unrolled for a plurality times in the loop unrolling section **113** to obtain the instruction schedule **107**. The number of times of development is set, for example, to be the least common multiple of B/D and N, when a size of a cache block which can be moved to the cache from the main memory by execution of the one prefetch instruction is defined as B, a size of the element referred by the memory reference instruction as D and an increment of the array reference element as N.

When a loop is unrolled, the redundant prefetch instructions are deleted subsequently from the instruction schedule **107** obtained by unrolling of the loop by the prefetch instruction deleting section **114**. Thereby, the final instruction schedule **103** not including the redundant prefetch instruction can be obtained. In regard to the deletion of the redundant prefetch instructions, since it is enough that the prefetch instruction is once issued for every B/D times, the other instructions are deleted so that the prefetch instruction is issued once every B/D times for each unrolled prefetch instruction.

In the method explained above, the number of times of loop unrolling is increased in some cases. Therefore, when it is required to keep low the number of times of the loop unrolling, for example, the loop is unrolled for an adequate number of times and the other instructions are deleted so that the prefetch instruction is generated in every other iteration of the B/D times. Thereby, a few redundant prefetch instructions may be issued, but an increase in the number of times of unrolling can be prevented.

Operations of the prefetch instruction deleting section **114** in the above explanation will now be explained with reference to the flowchart shown in FIG. **8**. First, in step **801**, a size of the cache block is set to a constant B, while a size of the reference object element as a constant D. In step **802**, whether unprocessed prefetch instructions remain or not is judged. When these instructions exist, the control is shifted to step **803** and when there is no such instructions, the processing is completed. In step **803**, the same unprocessed prefetch instructions are copied by the loop unrolling section **113** in FIG. 1 and are sequentially assigned to variables PFi ($0 \leqq i \leqq n$). In step **804***a*, if there exists a remainder: where i is divided by (B/D) (i mod (B/D))≠0) for $0 \leqq i \leqq n$, namely if i is not an integer multiple of B/D, the prefetch instruction PFi is deleted in step **804***b* and the control is shifted to step **802** to process the next prefetch instruction. Thereby, the prefetch instruction is issued once for every other iteration of multiples of B/D. The "MOD" function is recognized in PASCAL computer programming language to be the function that computes the remainder of division.

Next, an embodiment of scheduling the loop by method 3 will be explained. First, a prefetch instruction for the

memory reference instruction having a high possibility for generating a cache miss is generated from the intermediate language **102** as an input in the prefetch instruction generator **109** as in the case of method 1 to obtain the intermediate language **104** having the added prefetch instruction.

Next, a plurality of prefetch instructions generated by the prefetch instruction generator **109** are formed into groups and such groups are replaced with virtual prefetch instructions in the prefetch instruction replacing section **110**. In this replacement, for example, the virtual prefetch instructions for such a minimum integer number as is larger than M/(B/D) is generated when a size of a cache block which can be moved to the cache from the main memory by execution of one prefetch instruction is defined as B, a size of the element referred with the memory reference instruction as D and the number of prefetch instructions included in the intermediate language **104** as M and the prefetch instruction generated once for every other instruction of B/D number corresponds to one virtual prefetch instruction. When the virtual prefetch instruction is generated, the original prefetch instruction in the intermediate language **104** is deleted and a newly generated virtual prefetch instruction is added.

Next, the dependency graph generator **111** generates a dependency graph **105** from an input of the intermediate language **104**. In this case, unlike methods 1 and 2, there is not any dependent relationship between the virtual prefetch instruction and memory reference instruction. Subsequently, the software-pipelined instruction schedule **106** is obtained by applying the software pipelining to the loop in the software pipelining section **112**. Since the dependent relationship is not provided in method 3, unlike methods 1 and 2, between the prefetch instruction and corresponding memory reference instruction, a high degree of freedom for instruction array can be assured for application of the software pipelining.

Next, the software-pipelined instruction schedule **106** is unrolled for several times in the loop unrolling section **113** to obtain an instruction schedule **107**. The number of times of unrolling is set to the least common multiple, for example, of B/D and N, as in the case of method 2, when a size of the cache block which can be moved to the cache from the main memory by execution of one prefetch instruction is defined as B, a size of the element referred by the memory reference instruction as D and an increment of the array reference element as N. When the loop unrolling processing by the loop unrolling section **113** is completed, the virtual prefetch instruction included on the obtained instruction schedule **107** is recovered, in the prefetch instruction recovery section **115**, to the corresponding prefetch instruction replaced in the prefetch instruction replacing section **110**. To a certain virtual prefetch instruction VP, n prefetch instructions PF1, PF2, . . . ,PFn correspond, and when it is assumed that the virtual prefetch instruction VP is unrolled into m virtual prefetch instructions VP1, VP2, . . . ,VPm by the loop unrolling section **113**, such recovery processing is performed, for example, as explained hereunder.

In the case where n=B/D, when j=i mod(B/D), VPi is replaced with PFj.

In the case where n<B/D, when j=i mod(B/D), VPi is replaced with PFj, if $1 \leqq j \leqq n$, and VPi is deleted if n<j.

As a result, an instruction schedule **108** consisting of the original prefetch instruction can be obtained.

Next, an instruction schedule **103** not including redundant prefetch instructions can be obtained by adjusting the reference object address of the prefetch instruction of the instruction schedule **108** so that the data is prefetched, in the prefetch address adjusting section **116**, with the iteration which occurs sufficiently later for completion of data transfer by the prefetch instruction.

This address adjustment is performed as explained hereunder when the prefetch instruction, FETCH X[i], is issued, for example, for the array X.

That is, it is enough when the array element, which is referred to with the iteration for the number of times of a minimum integer which is equal to or larger than M/L+(B/D), where the number of cycles required for a single execution of the scheduled loop is defined as L and the number of cycles required for transfer of a cache block of the object data to the cache from the main memory with the prefetch instruction as M, is prefetched.

That is, when the number of times of iteration is defined as a, it is enough to adjust the reference address of the above prefetch instruction to FETCH X[i+α].

Hereafter, the processings executed by the prefetch instruction replacing section **110** and prefetch instruction recovery section **115** in method 3 will then be explained with reference to the flowchart.

FIG. **9** is an operation flowchart of the prefetch instruction replacing section **110** in FIG. **1**. First, in step **901**, a size of a cache block is set to a constant B, a size of the reference object element to a constant D and a value of variable n for recording the number of prefetch instructions to 0. In step **902**, it is determined whether or not any prefetch instructions still remain. When these instructions remain, the control skips to step **903**. When there is no such instruction, the processing is completed. In step **903**, whether a value of the variable n is 0 or not is judged. When the result is YES, the control skips to step **904**. If the result is NO, the control skips to step **906**. In step **904**, a new virtual prefetch instruction is generated and it is then stored in the variable VPF. In step **905**, the virtual prefetch instruction stored in the variable VPF is inserted into the intermediate language stream.

In step **906**, the prefetch instruction is selected and stored in the variable PF. In step **907**, the prefetch instruction recorded in the variable PF is provided to correspond to the virtual prefetch instruction recorded in the variable VPF. In step **908**, the prefetch instruction recorded in the variable PF is deleted from the intermediate language stream. In step **909**, the variable n shows an increment of 1 (one). In step **910**, whether the value of n is equal to (B/D) or not is judged. When the result is YES, the control skips to step **911**. When the result is NO, the control skips to step **902** to process the next prefetch instruction. In step **911**, the control skips to step **902** to process the next prefetch instruction. Thereby, the prefetch instruction is replaced once with the virtual prefetch instruction for every other (B/D) prefetches.

FIG. **10** is an operation flowchart of the prefetch instruction recovery section **115** in FIG. 1. First, in step **1001**, a size of the cache block is set to a constant B and a size of the reference object element as a constant D. In step **1002**, it is determined whether or not any virtual prefetch instructions still remain. When these instructions remain, the control skips to step **1003**. When there is no such instruction, the processing is completed. In step **1003**, the similar virtual prefetch instructions copied by the loop unrolling section **113** in FIG. 1 are sequentially stored in the variable VPi ($0 \leqq i \leqq m$). In step **1004**, the original prefetch instructions corresponding to VPi are stored in the variable PFj ($0 \leqq j < n$).

In step **1005**, whether the number n of prefetch instructions PFj is (B/D) or not is judged. When the result is YES, the control skips to step **1006** and when the result is NO, the control skips to step **1007**. In step **1006**, when j=i MOD(B/D) for each VPi, VPi is replaced with PFj and the control skips to step **1002** to process the next virtual prefetch instruction. In step **1007**, when j=i MOD(B/D) for each VPi, VPi is replaced with PFj if $0 \leqq j < n$, while VPi is deleted if $n \leqq j$ and the control skips to step **1002** to process the next virtual prefetch instruction. Thereby, the virtual prefetch instruction is recovered as the original prefetch instruction and each prefetch instruction is repeated once in every other B/D instructions.

FIG. **11** is an operation flowchart of the prefetch address adjusting section **116** in FIG. **1**. First, in step **1101**, a size of the cache block is set to a constant B, a size of the reference object element to a constant D, the number of execution cycles per loop to L, the number of cycles required for data transfer to the cache from the main memory to M and the number of iterations a to precedently issue the prefetch instruction to the minimum integer equal to or larger than M/L+(B/D). In step **1102**, it is determined whether or not any unprocessed prefetch instructions still remain. When these instructions remain, the control skips to step **1103**. When none of these remain, the processing is completed. In step **1103**, the unprocessed prefetches are selected and are stored in the variable PF. In step **1104**, the address referred by the prefetch instruction stored in the variable PF is changed to the address referred after the iteration of α times. Thereby, the prefetch instruction is issued sufficiently before generation of the memory reference instruction and the waiting time due to the memory reference can be hidden.

Subsequently, the effect of the scheduling by an embodiment of each method will be explained using practical examples. FIG. **12** is an example of a loop of the FORTRAN program used for explanation about an embodiment. The intermediate language shown in FIG. **13** can be constituted from the loop body of this program. An example of the schedule of the prefetch instruction in each method when such intermediate language is used as an input is indicated hereunder.

In the example of FIG. **13**, memory reference is performed with the instructions **1301**, **1302**, **1303**, but since the same address is referred to by the instructions **1301** and **1303**, one prefetch instruction is respectively generated for the arrays X and Y. In this example, a super-scalar type processor is assumed to execute in parallel the memory reference instruction, prefetch instruction and arithmetic instruction. However, the present invention can be applied not only to the super-scalar type processor but also to the sequential type processor and very long instruction word (VLIW) processor. In the following examples, it is assumed that the data to be used for the iteration of four times can be transferred to the cache with a single prefetch instruction and data transfer to the cache from the main memory requires **50** cycles.

Method 1

(1) Generation of the prefetch instructions

The prefetch instructions are generated for the arrays X and Y. The intermediate language having the added prefetch instructions is shown in FIG. **14**. In this figure, the instructions **1401** and **1402** are respectively prefetch instructions for the arrays X and Y.

(2) Generation of dependency graph

FIG. **15** illustrates a dependency graph for an intermediate language having the added prefetch instructions. In this figure, a node indicates an instruction and an arrow between the nodes indicates the dependent relationship. A numeral added to the right side of each arrow indicates the number of cycles for separating instructions. As shown in this figure, a dependent relationship having a delay of 50 cycles required for the transfer of data to the cache from the main memory is provided between the prefetch instruction **1501** for the array X and the load instruction **1503** for the array X; and between the prefetch instruction **1502** for the array Y and the load instruction **1504** for the array Y.

(3) Software pipelining

The software pipelining is applied to the dependency graph of FIG. **15**. The software-pipelined schedule is shown in FIG. **16**. The schedule shown in FIG. **16** is composed of a prologue section **1601** for initializing the loop, a kernel section **1602** for repeating the loop and an epilogue section **1603** for processing to terminate the loop. Each entry of FIG. **16** indicates the instruction slots corresponding to each entry. The prefetch instructions are assigned to the instruction slots **1604** and **1605** and are scheduled to be executed by the software pipelining in the iteration 10 times before the corresponding memory reference instruction. Since the schedule which satisfies dependent relationship between instructions has been obtained by the software pipelining, the waiting time generated by the memory reference can be eliminated.

Method 2

In the embodiment of method 1, two prefetch instructions are generated for a single iteration. Since the data used for an iteration of four times can be prefetched with the prefetch instruction, it is useless to issue the prefetch instruction for each processing. Therefore, the generation of useless prefetch instructions can be controlled, in method 2, by applying the following processing to method 1.

(4) Loop development

The kernel section of the software-pipelined loop constituted in item (3) of method (3) is unrolled. Since it is assumed, in this embodiment, that the data referred to by the iteration of four times can be transferred to the cache with a single prefetch operation, it is enough when the prefetch instruction is generated once for every iteration of four times. Therefore, the schedule indicated in FIG. **17** can be obtained by unrolling the kernel section four times. The schedule shown in FIG. **17** is composed of the prologue section **1701**, unrolled kernel section **1702** and epilogue section **1703**. The prefetch instruction for the array X in the kernel section **1702** is unrolled to the instruction slots **1704**, **1706**, **1708** and **1710**, while the prefetch instruction for the array Y is unrolled to the instruction slots **1705**, **1707**, **1709** and **1711**.

(5) Deletion of redundant prefetch instructions

The redundant prefetch instructions for the arrays X and Y are deleted so that one prefetch instruction is generated for every iteration of four times for the instruction schedule of FIG. **17** obtained by development of the loop. Thereby, generation of useless prefetch instructions can be controlled and the schedule shown in FIG. **18** can be obtained. In FIG. **18**, the redundant prefetch instructions **1805**, **1806**, **1807**, **1808**, **1810** and **1811** are deleted by the prologue section **1802** and the data for iteration of four times can be effectively prefetched by the respective prefetch instructions of the instruction slot **1804** for the array X and the instruction slot **1809** for the array Y.

Method 3

In method 3, the prefetch instructions are scheduled as explained hereunder, considering that useless prefetch instructions are never issued.

(1) Generation of prefetch instructions

The generation of the prefetch instructions is executed in the same manner as for method 1.

(2) Replacement of prefetch instruction and generation of dependency graph

A plurality of prefetch instructions generated in item (1) explained above are grouped to form virtual prefetch instructions in order to constitute a dependency graph. The obtained dependency graph is shown in FIG. **19**. As shown in FIG. **19**, the prefetch instruction **1901** for the array X and the prefetch instruction **1902** for the array Y are combined and are then replaced with the virtual prefetch instruction **1903**. Unlike methods 1 and 2, no dependent relationship is provided, in method **3**, between the virtual prefetch instruction and the corresponding memory reference instruction.

(3) Software pipelining

The software pipelining is applied to the loop body to which the virtual prefetch instruction is added. As a result, the software-pipelined schedule can be obtained as shown in FIG. **20**. The schedule shown in FIG. **20** is composed of the prologue section **2001**, kernel section **2002** and epilogue section **2003**. To the instruction generating slot **2004** of the kernel section **2002**, the virtual prefetch is scheduled.

(4) Loop development

Like the case of method 2, the kernel section of the software-pipelined schedule constituted in item (3) explained above is unrolled for four times. Thereby, the schedule shown in FIG. **21** can be obtained. The schedule shown in FIG. **21** is composed of the prologue section **2101**, unrolled kernel section **2102** and epilogue section **2103**. The virtual prefetch instruction scheduled in the kernel section **2103** by the loop unrolling section is copied into the instruction generating slots **2104**, **2105**, **2106** and **2107**.

(5) Recovery of prefetch address

The virtual prefetch instruction unrolled for the instruction slots **2104**, **2105**, **2106** and **2107** of the kernel section **2102** of FIG. **21** is replaced with the original prefetch instruction. The result is shown in FIG. **22**. Since the virtual prefetch instruction unrolled for the instruction slots **2104**, **2105**, **2106** and **2107** of FIG. **21** is obtained by replacing the prefetch instruction for the arrays X and Y, the prefetch instruction is inserted into the instruction slots **2204** and **2206** of FIG. **22** so that the prefetch instruction for respective array is generated once for every iteration of four times. In this case, since the number of original prefetch instructions is less than the number of times of iteration for making reference to the data which can be transferred to the cache from the main memory with only a single prefetch operation, the instruction slots **2205** and **2207** for the unrolled virtual prefetch instruction are maintained as the idle slots.

(6) Adjustment of prefetch address

The address as an object of the prefetch is adjusted so that an interval as great as the number of cycles sufficient for termination of the transfer of a cache block to the cache from the main memory is maintained between issuance of the prefetch instruction and the issuance of the instruction for making reference to the data transferred to the cache by the prefetch instruction. Since the cycle required for the transfer of the cache block to the cache from the main memory is 50 cycles and four cycles are required for a single iteration, the reference destination of a prefetch instruction is changed here so that the data referred to after iterations of 14 times is prefetched as shown in FIG. 22.

As explained above, the instruction schedule **103** including the prefetch instruction can be generated by the scheduler **101** using the intermediate language of FIG. **1** as an input. That is, when the reference to data is not continuously performed in the iteration of the loop, since the memory reference instruction corresponding to the prefetch instruction can be issued with an interval as long as the number of cycles required for the transfer of data to the cache from the main memory by utilizing method 1, the waiting time due to the memory reference can be hidden. Moreover, when the reference to data is continuously performed, issuance of the redundant prefetch instructions can be controlled by utilizing methods 2 and 3. In addition, since the dependent relationship is not provided, in comparison with method 2, between the virtual prefetch instruction and the memory reference instruction in method 3, the degree of freedom for the array of instructions increases and since the software-pipelining is applied considering the development of the kernel section, generation of delay time due to the dependent relationship between the instructions can be kept at a minimum.

Although the embodiments of the invention have been described in relation to schedulers, sections, generators, unrolling sections and the like, it is understood that these components of the invention are embodied by software stored in computer memory or on memory storage media for execution by a computer and that the software is executed to enable the methods to be performed by a computer, such as a general purpose computer.

According to the present invention, the waiting time due to memory reference, etc. during execution of the programs can be reduced by effectively scheduling the prefetch instruction. Thereby, the present invention is very effective for high speed execution of the computer programs.

Namely, according to the present invention, if reference to the memory is not performed continuously, the software pipelining can be applied by method 1 keeping sufficiently long intervals between the prefetch instructions and the memory reference instructions. Moreover, when a reference to the memory is performed continuously, the instruction is deleted after application of the software pipelining by method 2 or a plurality of prefetch instructions are replaced with the virtual prefetch instructions by method 3 for application of the software pipelining and thereafter such virtual prefetch instructions are recovered to the original prefetch instructions to control the issuance of the useless prefetch instructions in view of realizing the effective schedule.

We claim:

1. A data prefetch method in a compiler for compiling programs to be executed on a computer having a prefetch instruction for transferring data to a cache memory from a main memory in parallel with execution of other instructions, comprising:

(a) converting a source program in a loop of a program into intermediate code;

(b) replacing a plurality of prefetch instructions included in a loop of a program into one virtual prefetch instruction independent of memory reference;

(c) generating a dependency graph having edges and showing a required delay between said virtual prefetch instruction and an instruction for memory reference in accordance with said intermediate code that is longer than a time required to transfer the data of the virtual prefetch instruction to the cache memory from the main memory;

(d) executing instruction scheduling by applying software pipelining, for scheduling instructions to hide latency between instructions by execution through overlap of different iterations of the loop, to said dependency graph; and

(e) unrolling the obtained schedule for a plurality of times to replace said unrolled virtual prefetch instruction with a plurality of initial prefetch instructions.

2. A data prefetch method according to claim 1, wherein said step (e) further comprises adjusting the address which

is referred to by said replaced prefetch instruction to the address which is referred to by the iteration which is sufficiently later to complete the data transfer by said prefetch instruction.

3. A data prefetch method in a compiler for compiling programs to be executed on a computer having a prefetch instruction for transferring data to a cache memory from a main memory in parallel with execution of the other instructions, comprising:

(a) converting a source program in a loop of a program into intermediate code;

(b) replacing a plurality of prefetch instructions included in a loop of a program into one virtual prefetch instruction independent of memory reference;

(c) generating a dependency graph having edges and showing a required delay between said virtual prefetch instruction and an instruction for memory reference in accordance with said intermediate code that is longer than a time required to transfer the data of the virtual prefetch instruction to the cache memory from the main memory;

(d) executing instruction scheduling by applying a software pipelining, for scheduling instructions to hide latency between instructions by execution through overlap of different iterations of the loop, to said dependency graph; and

(e) unrolling said instruction scheduling,

wherein said step (e) further comprises:

(e1) unrolling the obtained schedule for a plurality of times; and

(e2) replacing said unrolled virtual prefetch instruction with a plurality of initial prefetch instructions.

4. A data prefetch method according to claim 3, wherein said step (e) further comprises adjusting the address which is referred to by said replaced prefetch instruction to the address which is referred to by the iteration which is sufficiently later to complete the data transfer by said prefetch instruction.

5. A compile program stored on a computer readable storage medium executing a data prefetch method on a computer having a prefetch instruction for transferring data to a cache memory from main memory in parallel with execution of the other instructions, comprising:

(a) converting a source program in a loop of a program into intermediate code;

(b) replacing a plurality of prefetch instructions included in a loop of a program into one virtual prefetch instruction independent of memory reference;

(c) generating a dependency graph having edges and showing a required delay between said virtual prefetch instruction and an instruction for memory reference in accordance with said intermediate code that is longer than a time required to transfer the data of the prefetch instruction to the cache memory from the main memory;

(d) executing instruction scheduling by applying a software pipelining, for scheduling instructions to hide latency between instructions by execution through overlap of different iterations of the loop, to said dependency graph; and

(e) unrolling said instruction scheduling,

wherein said step (e) further comprises:

(e1) unrolling the obtained schedule for a plurality of times; and

(e2) replacing said unrolled virtual prefetch instruction with a plurality of initial prefetch instructions.

6. A compile program according to claim 5, wherein said step (e) further comprises adjusting the address which is referred to by said replaced prefetch instruction to the address which is referred to by the iteration which is

sufficiently later to complete the data transfer by said prefetch instruction.

* * * * *

US005862385A

United States Patent [19]
Iitsuka

[11] **Patent Number: 5,862,385**
[45] **Date of Patent: Jan. 19, 1999**

[54] **COMPILE METHOD FOR REDUCING CACHE CONFLICT**

[75] Inventor: **Takayoshi Iitsuka**, Sagamihara, Japan

[73] Assignee: **Hitachi, Ltd.**, Japan

[21] Appl. No.: **861,187**

[22] Filed: **May 21, 1997**

Related U.S. Application Data

[63] Continuation of Ser. No. 303,007, Sep. 8, 1994.

[30] **Foreign Application Priority Data**

Sep. 10, 1993 [JP] Japan 5-250014

[51] **Int. Cl.^{6}** **G06F 9/45**
[52] **U.S. Cl.** **395/709**; 395/707; 395/708
[58] **Field of Search** 395/701, 703, 395/704, 705, 706, 707, 708, 709, 445, 460, 461

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,142,634	8/1992	Fite et al.	395/375
5,337,415	8/1994	DeLane et al.	395/375
5,442,790	8/1995	Nosenchuck	395/700
5,481,751	1/1996	Alpert et al.	395/800
5,488,729	1/1996	Vegesna et al.	395/800
5,497,499	3/1996	Garg et al.	395/800

OTHER PUBLICATIONS

Jouppi, N., "Improved Direct–Mapped Cache Performance by the Addition of a Small Fully–Associative Cache and Prefetch Buffers", Computer Architecture, 1990 International Symposium, IEEE, pp. 363–373, May 1990.

Gosmann et al, "Code Reorganization for Instruction Caches", System Sciences, 1993 Annual Hawaii Int'l Conference IEEE pp. 214–223, Jan. 1993.

Pyster, A., "Compiler Design and Construction", 1980, pp. 11, Jan. 1990.

Leonard, D, "The Moral Dilemma of Decompilers", Data Based Advisor, v10 n8 p. 38(3), Aug. 1992.

Aho, Alfred V. et al., *Compilers* (©1985, Addison Wesley Publishing Co.), pp. 464–473.

Hennessy, J. et al., *Computer Architecture: A Quantitative Approach* (©1990 Morgan Kaufmann Publishers, Inc., Palo Alto, California), pp. 408–425.

Krishnamurthy, Sanjay M., "A Brief Survey of Papers on Scheduling for Pipelined Processors", *ACM SIGPLAN Notices*, vol. 25, No. 7, Jul. 1990, pp. 97–106.

Primary Examiner—James P. Trammell
Assistant Examiner—Kakali Chaki
Attorney, Agent, or Firm—Michaelson & Wallace; Peter L. Michaelson; John C. Pokotylo

[57] **ABSTRACT**

A compiling method, for use with programs to be executed on a computer with cache memory, which programs would otherwise generate decreased performance due to cache conflicts arising from conflicting cache access(es), for reducing the generation of such cache conflict(s) by reordering the order of memory reference code in the program such that conflicting memory references do not start before the completion of memory accesses to a memory block.

6 Claims, 30 Drawing Sheets

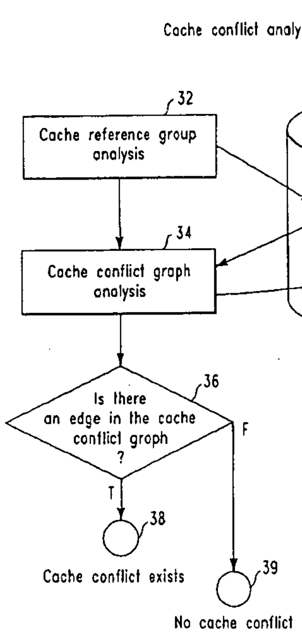

Cache conflict analysis
30
32
Cache reference group analysis
9
Cache conflict information
9900
Cache reference group information
34
Cache conflict graph analysis
9500
Cache conflict graph
36
Is there an edge in the cache conflict graph ?
F
T
38
Cache conflict exists
39
No cache conflict

FIG. 1

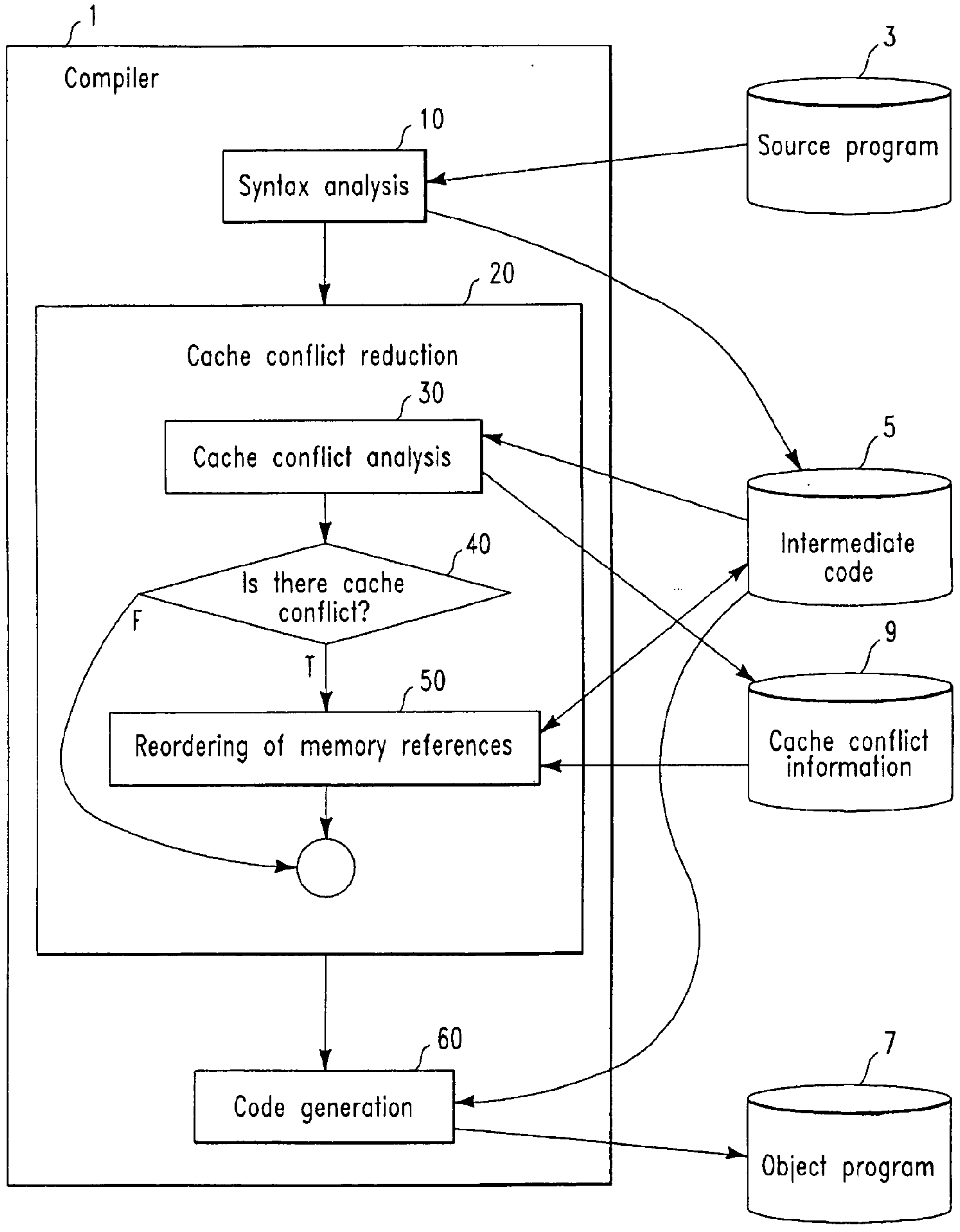

1
Compiler
10
Syntax analysis
20
Cache conflict reduction
30
Cache conflict analysis
40
Is there cache conflict?
F
T
50
Reordering of memory references
60
Code generation
3
Source program
5
Intermediate code
9
Cache conflict information
7
Object program

FIG. 2 (PRIOR ART)

Source Program

```
      COMMON /BLK1/A,B,C
      REAL A(256,128),B(256,128,3),C(256,128)
      DO 10 J=1,128
3     DO 20 I=1,255
      C(I,J) = A(I,J) * B(I,J,2) + A(I+1,J) * B(I,J,3) + B(I,J,1)
   20 CONTINUE
   10 CONTINUE
```

FIG. 3 (PRIOR ART)

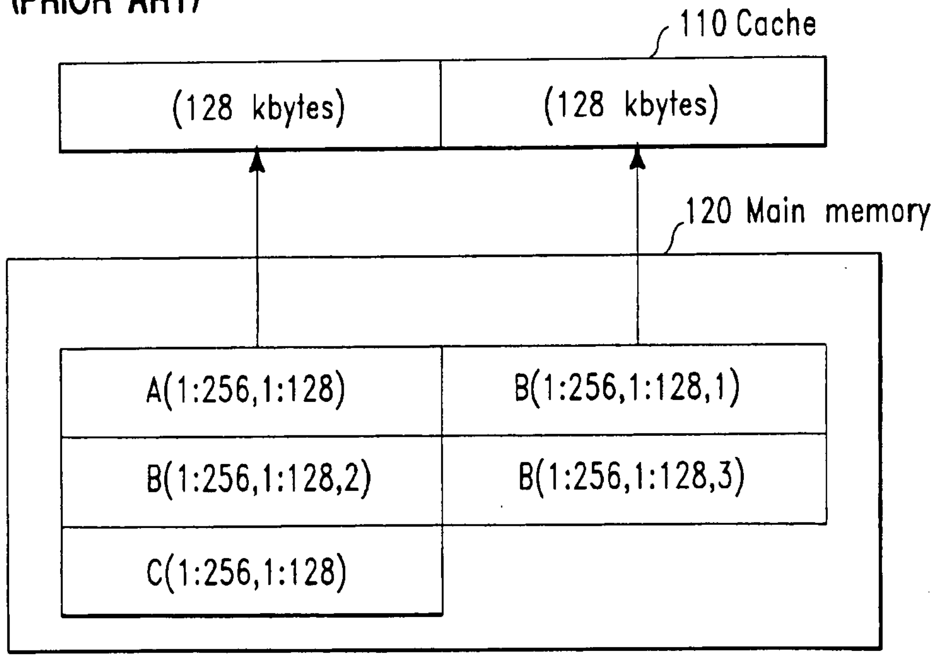


FIG. 4 (PRIOR ART)

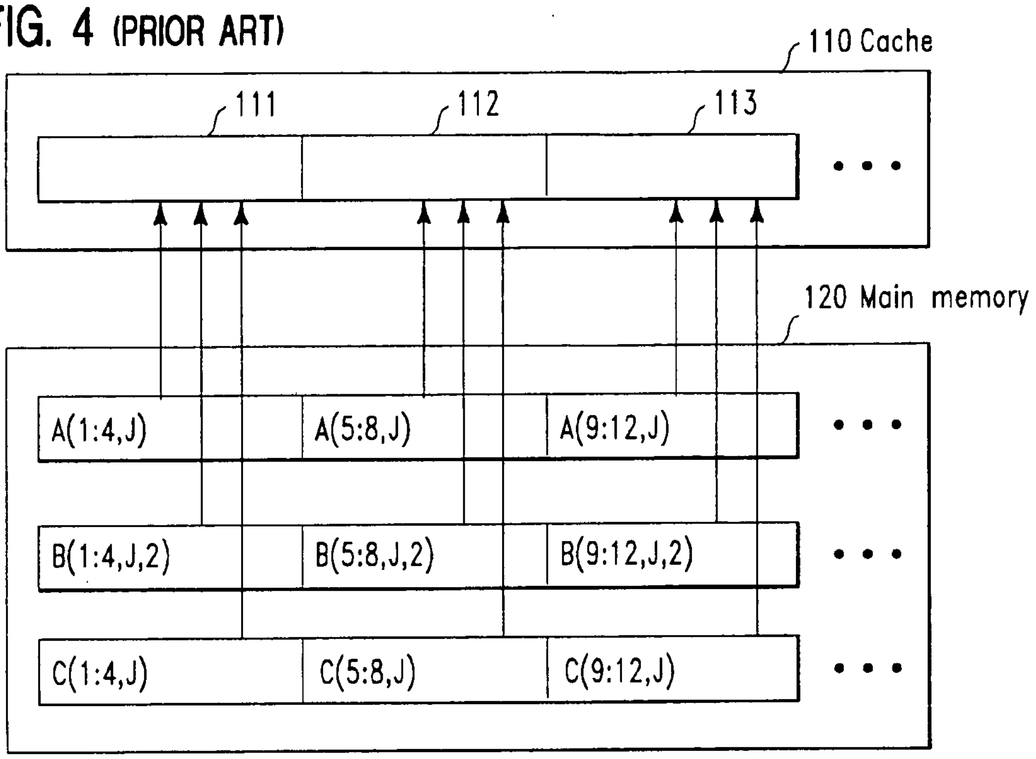


FIG. 5 (PRIOR ART)

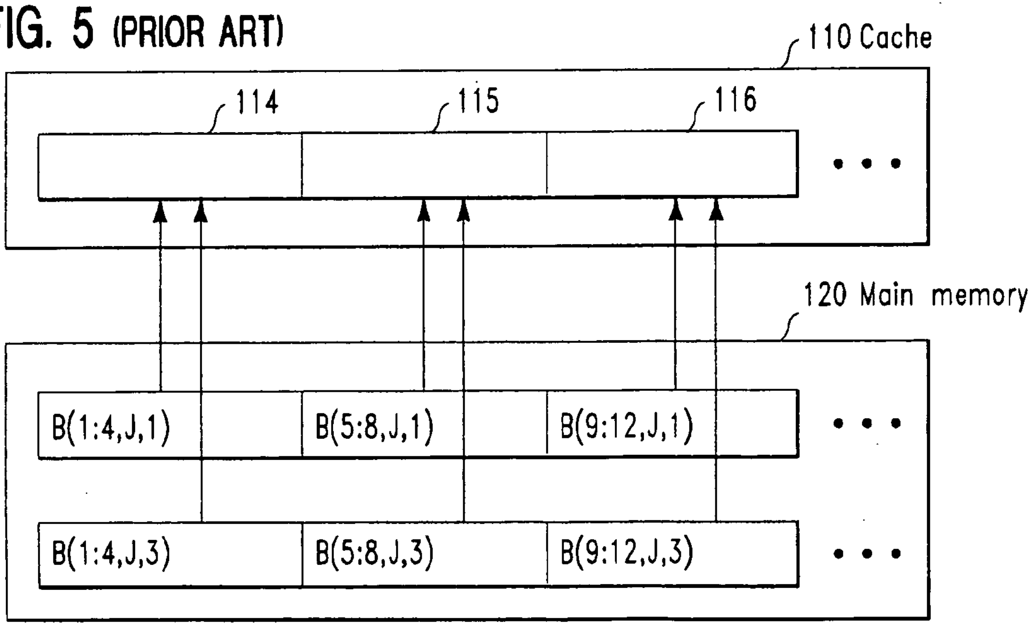


FIG. 6 (PRIOR ART)

Object Program

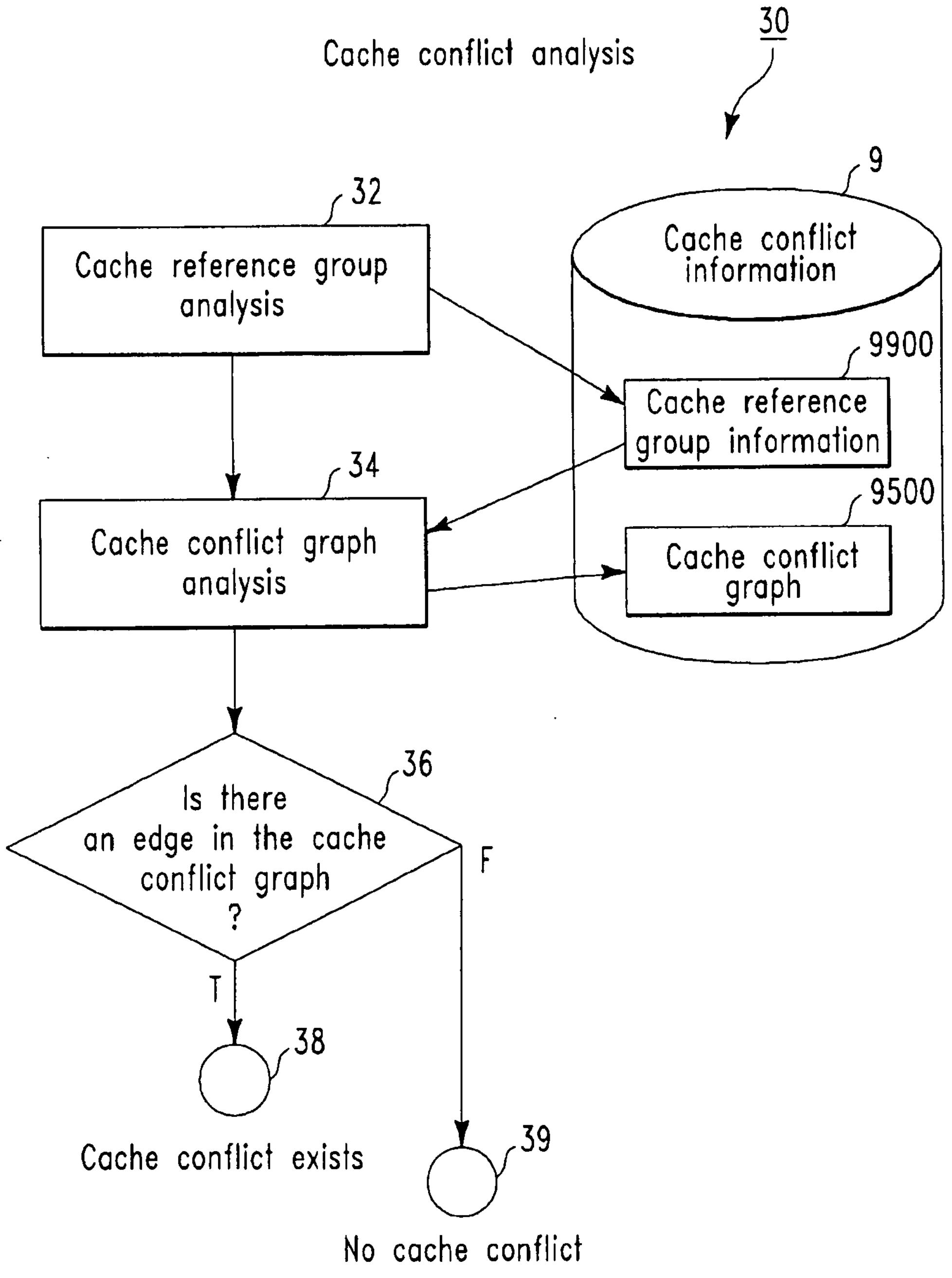
7

61	62	63	64	65
Instruction number	Label	Instruction	Operand	Processing contents
1	loop:	fload	A(r1,J),fr1	fr1 = A(I,J)
2		fload	B(r1,J,2),fr2	fr2 = B(I,J,2)
3		fmpy	fr1,fr2,fr2	fr2 = fr1 * fr2
4		fload	A(r1+1,J),fr3	fr3 = A(I+1,J)
5		fload	B(r1,J,3)fr4	fr4 = B(I,J,3)
6		fmpy	fr3,fr4,fr4	fr4 = fr3 * fr4
7		fadd	fr2,fr4,fr4	fr4 = fr2 + fr4
8		fload	B(r1,J,1),fr5	fr5 = B(I,J,1)
9		fadd	fr4,fr5,fr5	fr5 = fr4 + fr5
10		fstore	fr5,C(r1,J)	C(I,J) = fr5
11		addi	r1,#1,r1	r1 = r1 + 1
12		cmp	r1,#255	compare r1 and 255
13		ble	loop	if r1 ≦ 255 goto loop

FIG. 7 (PRIOR ART)

70	71	72	73	74	75	76
I value	Instruction number	Referenced array element	Hit status	Accepted block	Rejected block	Cache miss caused by cache conflict
I = 1	1	A(1,J)	Miss	A(1:4,J)		
	2	B(1,J,2)	Miss	B(1:4,J,2)	A(1:4,J)	
	4	A(2,J)	Miss	A(1:4,J)	B(1:4,J,2)	YES
	5	B(1,J,3)	Miss	B(1:4,J,3)		
	8	B(1,J,1)	Miss	B(1:4,J,1)	B(1:4,J,3)	
	10	C(1,J)	Miss	C(1:4,J)	A(1:4,J)	
I = 2	1	A(2,J)	Miss	A(1:4,J)	C(1:4,J)	YES
	2	B(2,J,2)	Miss	B(1:4,J,2)	A(1:4,J)	YES
	4	A(3,J)	Miss	A(1:4,J)	B(1:4,J,2)	YES
	5	B(2,J,3)	Miss	B(1:4,J,3)	B(1:4,J,1)	YES
	8	B(2,J,1)	Miss	B(1:4,J,1)	B(1:4,J,3)	YES
	10	C(2,J)	Miss	C(1:4,J)	A(1:4,J)	YES
I = 3	1	A(3,J)	Miss	A(1:4,J)	C(1:4,J)	YES
	2	B(3,J,2)	Miss	B(1:4,J,2)	A(1:4,J)	YES
	4	A(4,J)	Miss	A(1:4,J)	B(1:4,J,2)	YES
	5	B(3,J,3)	Miss	B(1:4,J,3)	B(1:4,J,1)	YES
	8	B(3,J,1)	Miss	B(1:4,J,1)	B(1:4,J,3)	YES
	10	C(3,J)	Miss	C(1:4,J)	A(1:4,J)	YES
I = 4	1	A(4,J)	Miss	A(1:4,J)	C(1:4,J)	YES
	2	B(4,J,2)	Miss	B(1:4,J,2)	A(1:4,J)	YES
	4	A(5,J)	Miss	A(5:8,J)		
	5	B(4,J,3)	Miss	B(1:4,J,3)	B(1:4,J,1)	YES
	8	B(4,J,1)	Miss	B(1:4,J,1)	B(1:4,J,3)	YES
	10	C(4,J)	Miss	C(1:4,J)	A(1:4,J)	YES
			⋮			

FIG. 8


Cache conflict analysis
30
9
Cache conflict information
32
Cache reference group analysis
9900
Cache reference group information
34
Cache conflict graph analysis
9500
Cache conflict graph
36
Is there an edge in the cache conflict graph ?
F
T
38
Cache conflict exists
39
No cache conflict

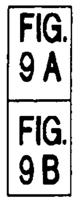
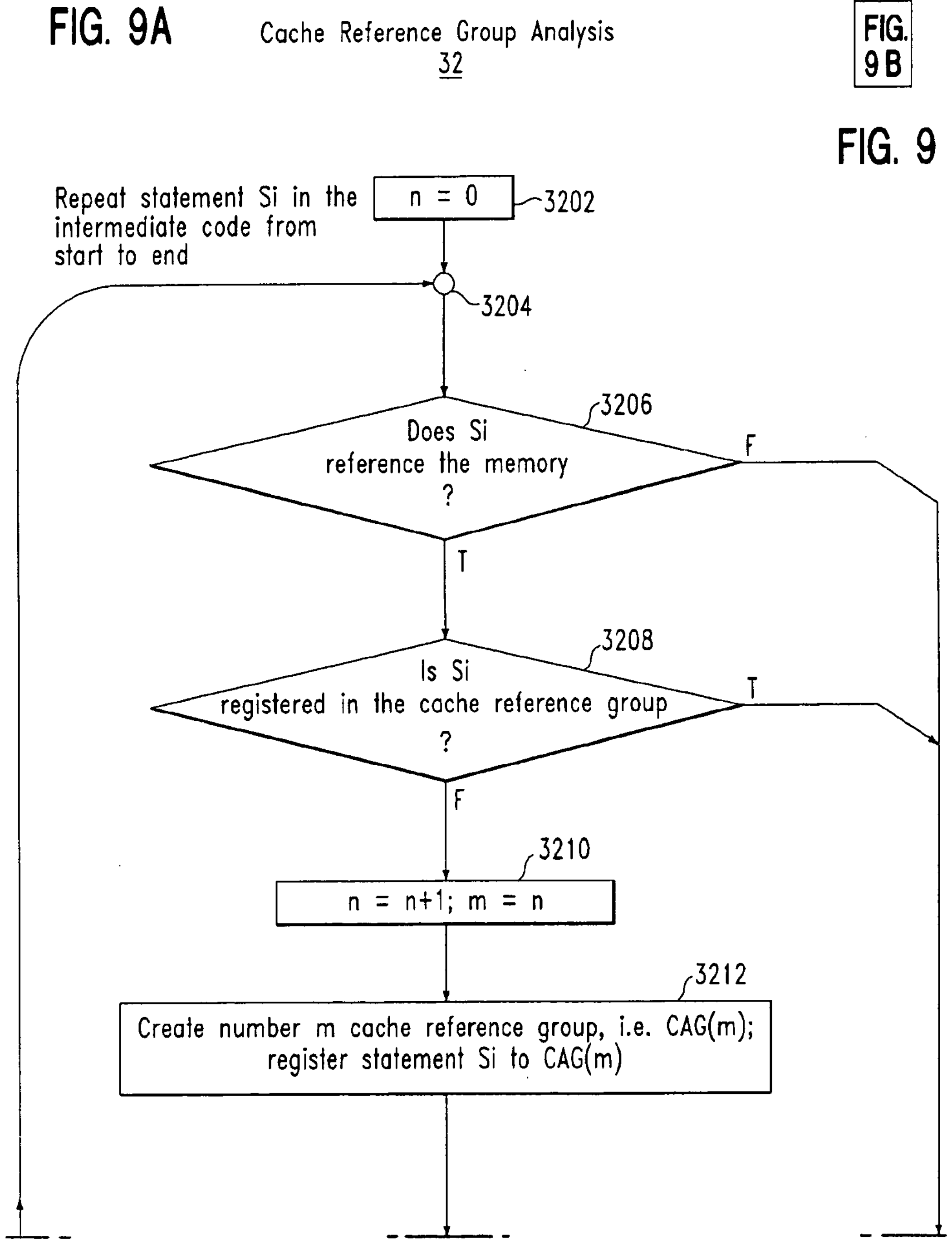


FIG. 9A
FIG. 9B

FIG. 9

FIG. 9A


Cache Reference Group Analysis
32
Repeat statement Si in the intermediate code from start to end
n = 0
3202
3204
Does Si reference the memory ?
3206
F
T
Is Si registered in the cache reference group ?
3208
T
F
n = n+1; m = n
3210
Create number m cache reference group, i.e. CAG(m); register statement Si to CAG(m)
3212

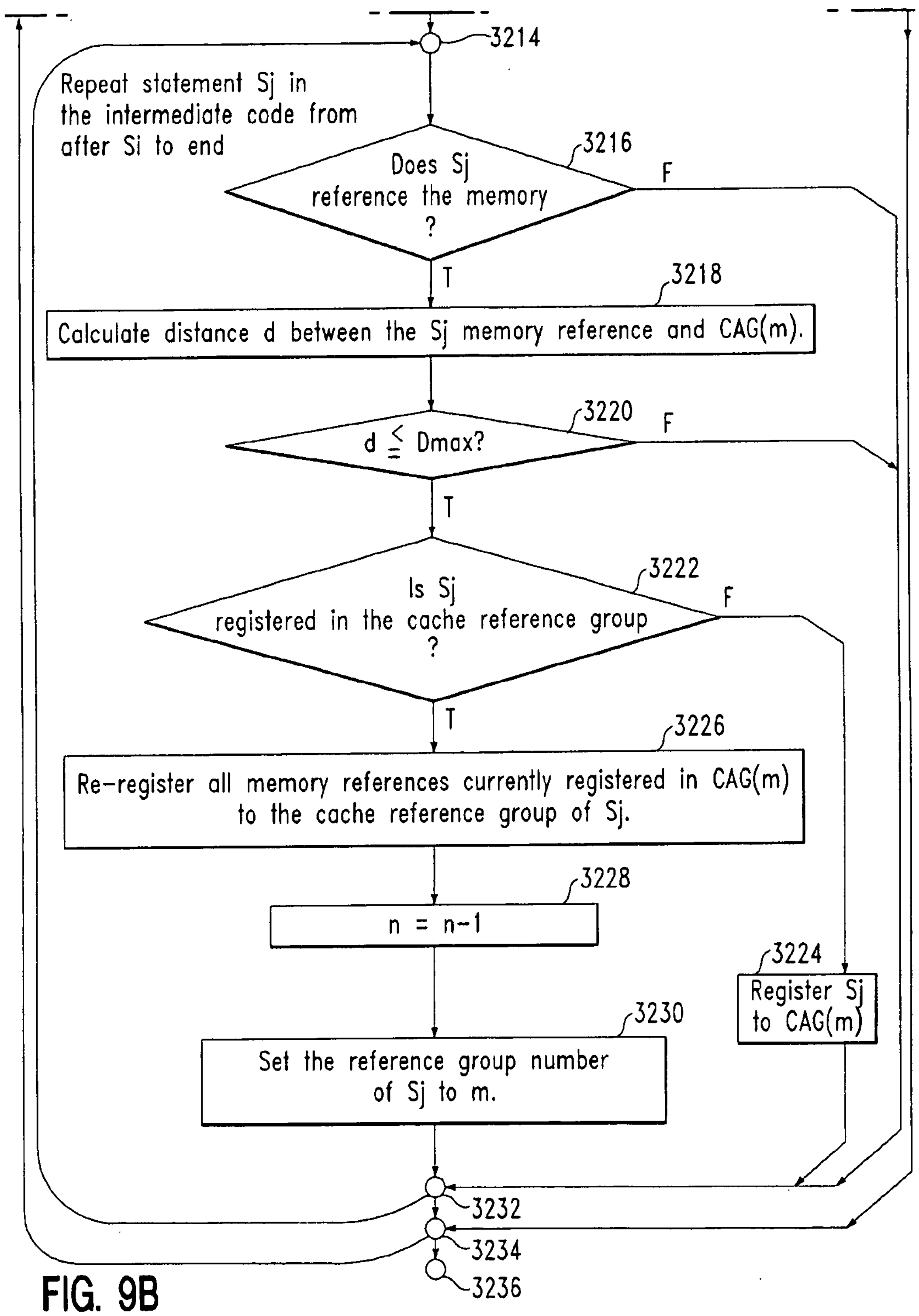

3214
Repeat statement Sj in the intermediate code from after Si to end
Does Sj reference the memory ?
3216
F
T
3218
Calculate distance d between the Sj memory reference and CAG(m).
d ≤ Dmax?
3220
F
T
Is Sj registered in the cache reference group ?
3222
F
T
3226
Re-register all memory references currently registered in CAG(m) to the cache reference group of Sj.
3228
n = n-1
3224
Register Sj to CAG(m)
3230
Set the reference group number of Sj to m.
3232
3234
3236

FIG. 9B

FIG. 10

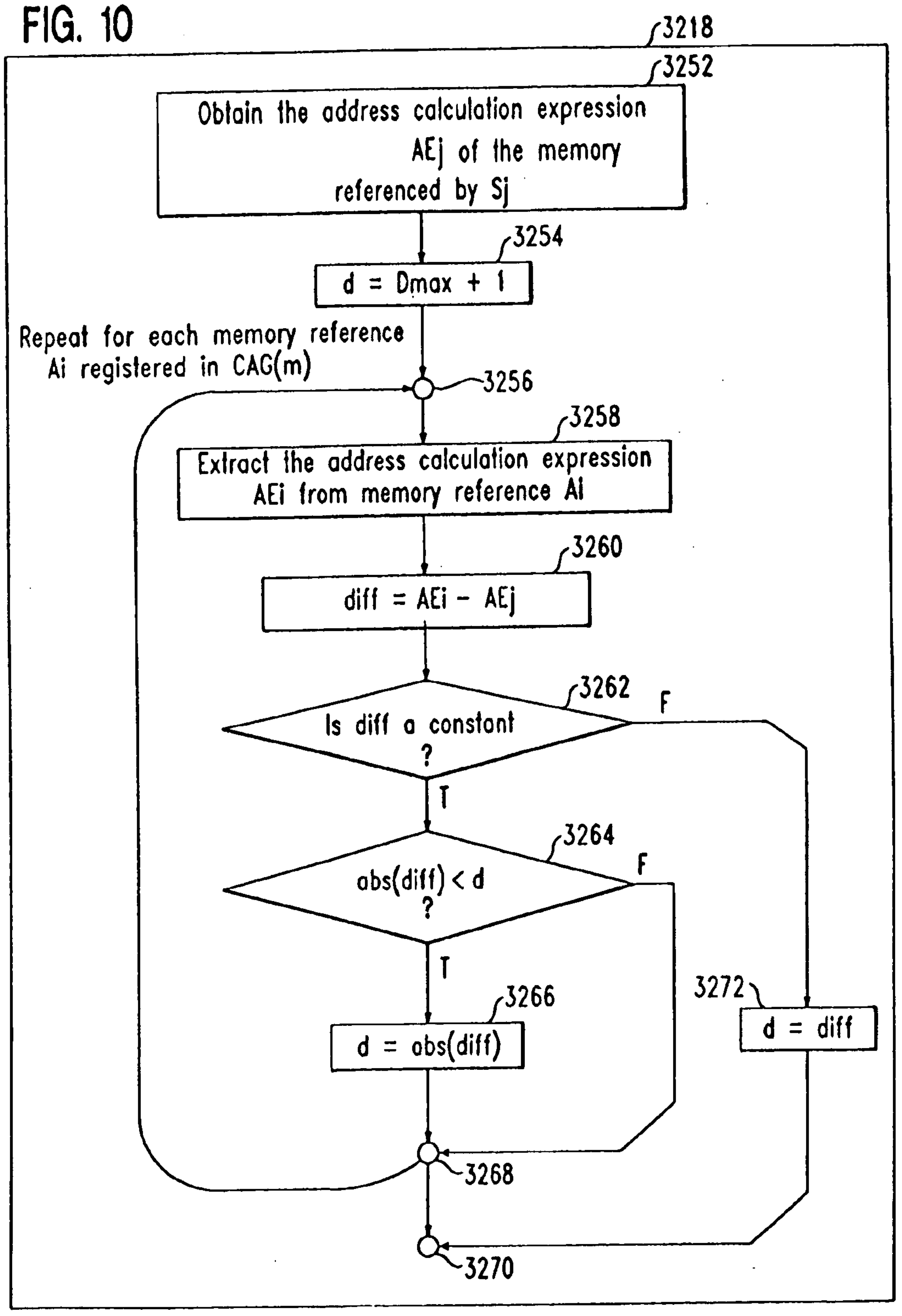

3218
3252
Obtain the address calculation expression AEj of the memory referenced by Sj
3254
d = Dmax + 1
Repeat for each memory reference Ai registered in CAG(m)
3256
3258
Extract the address calculation expression AEi from memory reference Ai
3260
diff = AEi − AEj
3262
Is diff a constant ?
F
T
3264
abs(diff) < d ?
F
T
3266
d = abs(diff)
3272
d = diff
3268
3270

FIG. 11

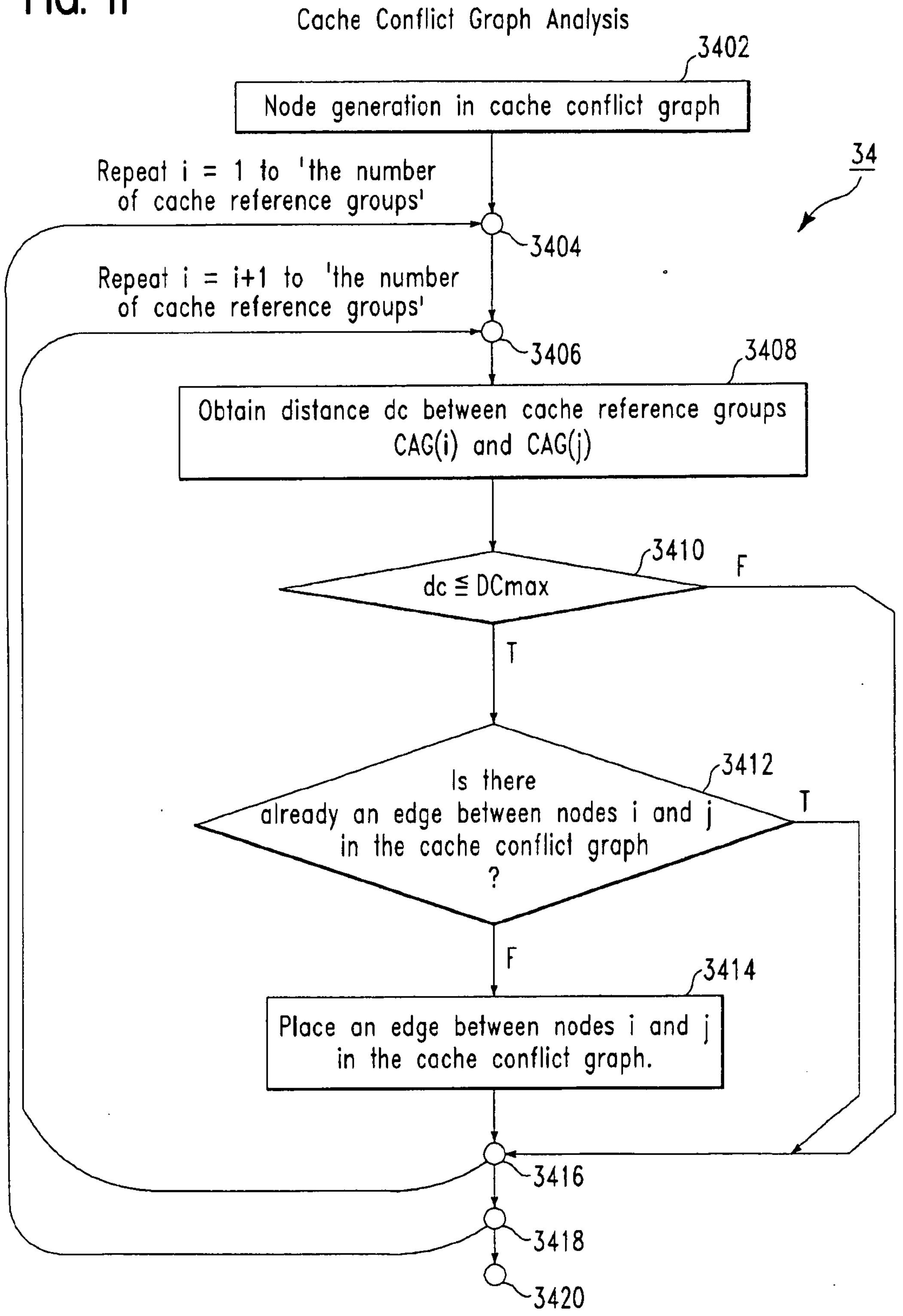

Cache Conflict Graph Analysis
3402
Node generation in cache conflict graph
34
Repeat i = 1 to 'the number of cache reference groups'
3404
Repeat i = i+1 to 'the number of cache reference groups'
3406
3408
Obtain distance dc between cache reference groups CAG(i) and CAG(j)
3410
dc ≦ DCmax
F
T
3412
Is there already an edge between nodes i and j in the cache conflict graph ?
T
F
3414
Place an edge between nodes i and j in the cache conflict graph.
3416
3418
3420

FIG. 12

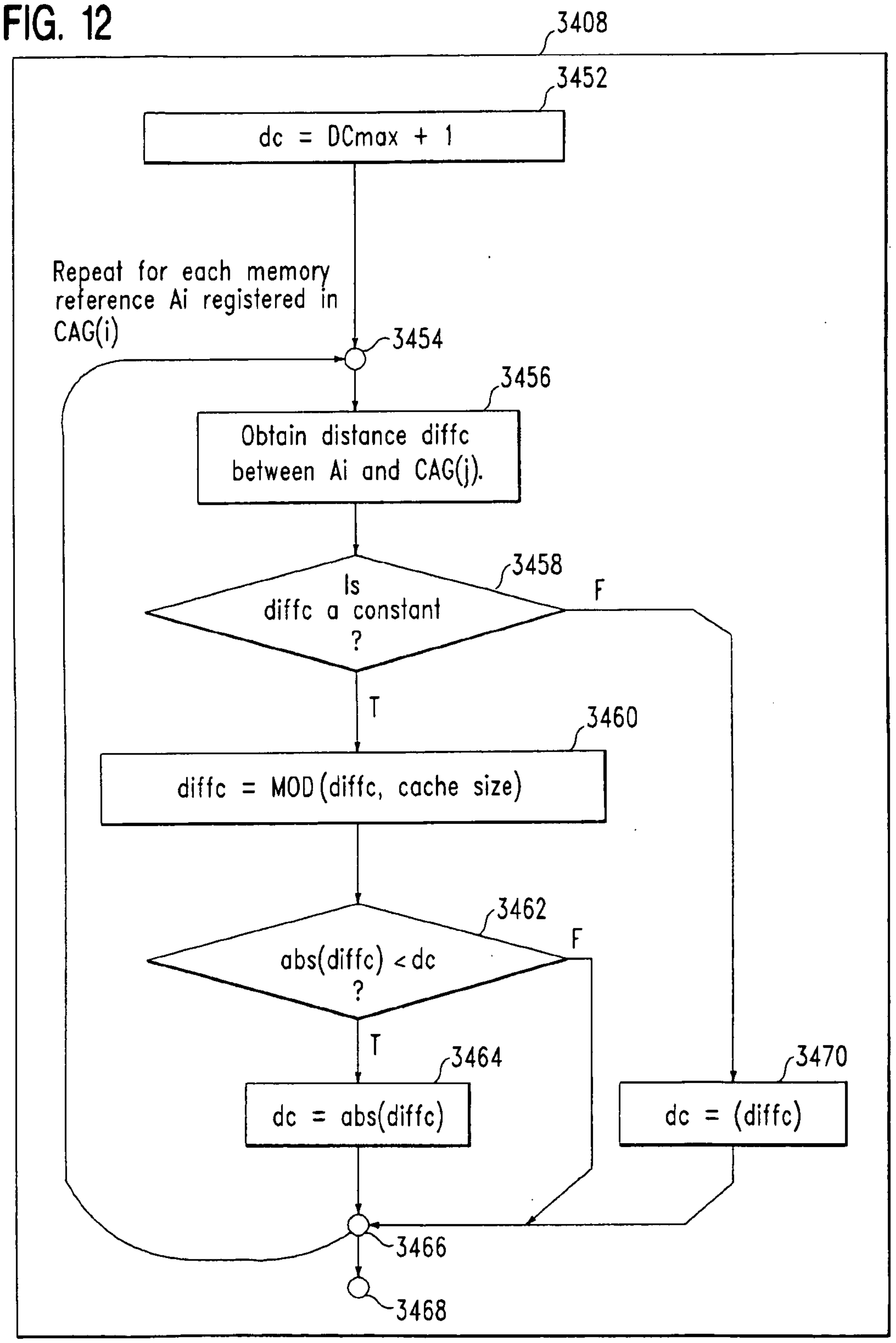

3408
3452
dc = DCmax + 1
Repeat for each memory reference Ai registered in CAG(i)
3454
3456
Obtain distance diffc between Ai and CAG(j).
3458
Is diffc a constant ?
F
T
3460
diffc = MOD(diffc, cache size)
3462
abs(diffc) < dc ?
F
T
3464
dc = abs(diffc)
3470
dc = (diffc)
3466
3468

FIG. 13

Reordering of Memory References

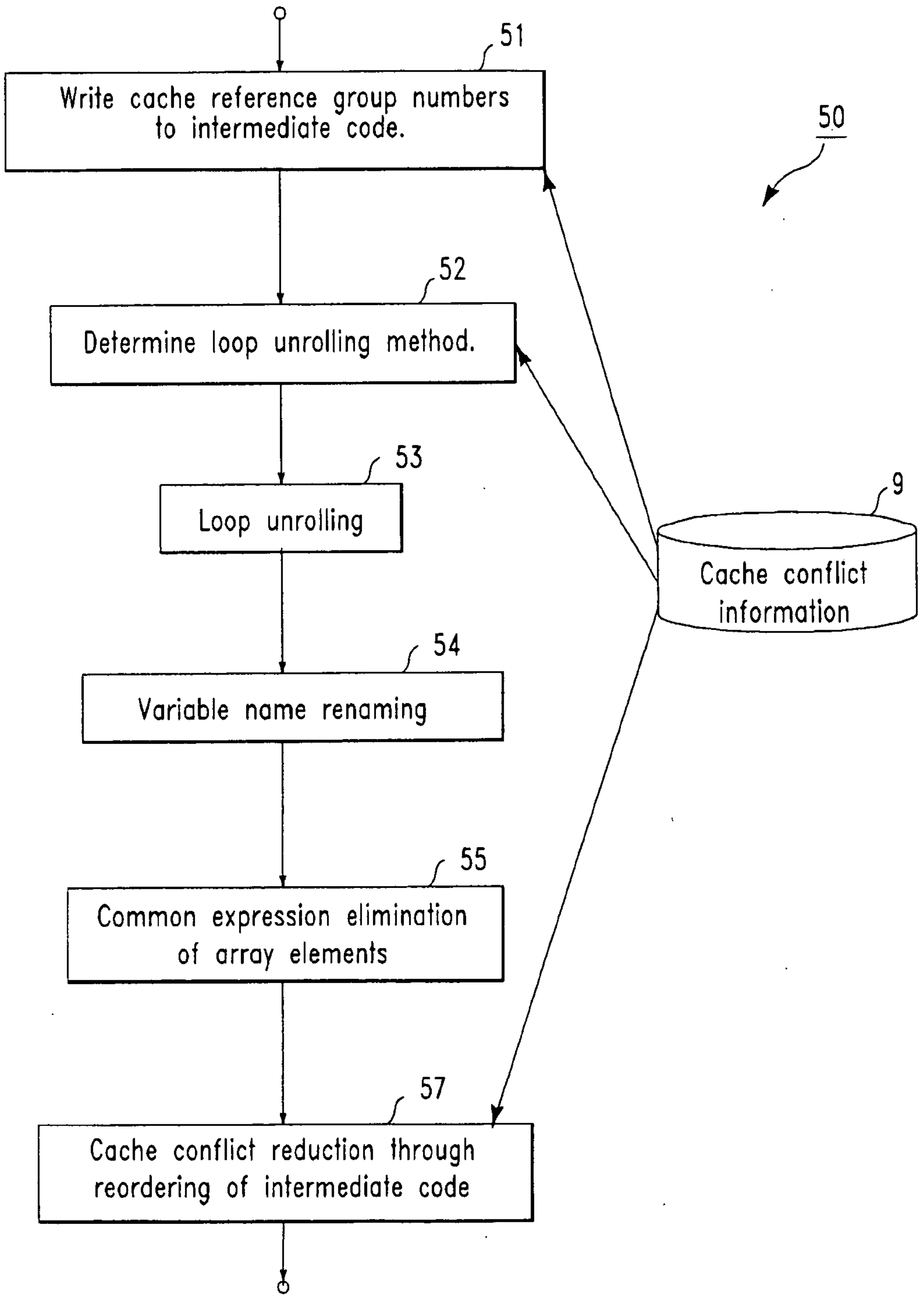

51
Write cache reference group numbers to intermediate code.
50
52
Determine loop unrolling method.
53
Loop unrolling
9
Cache conflict information
54
Variable name renaming
55
Common expression elimination of array elements
57
Cache conflict reduction through reordering of intermediate code

FIG. 14

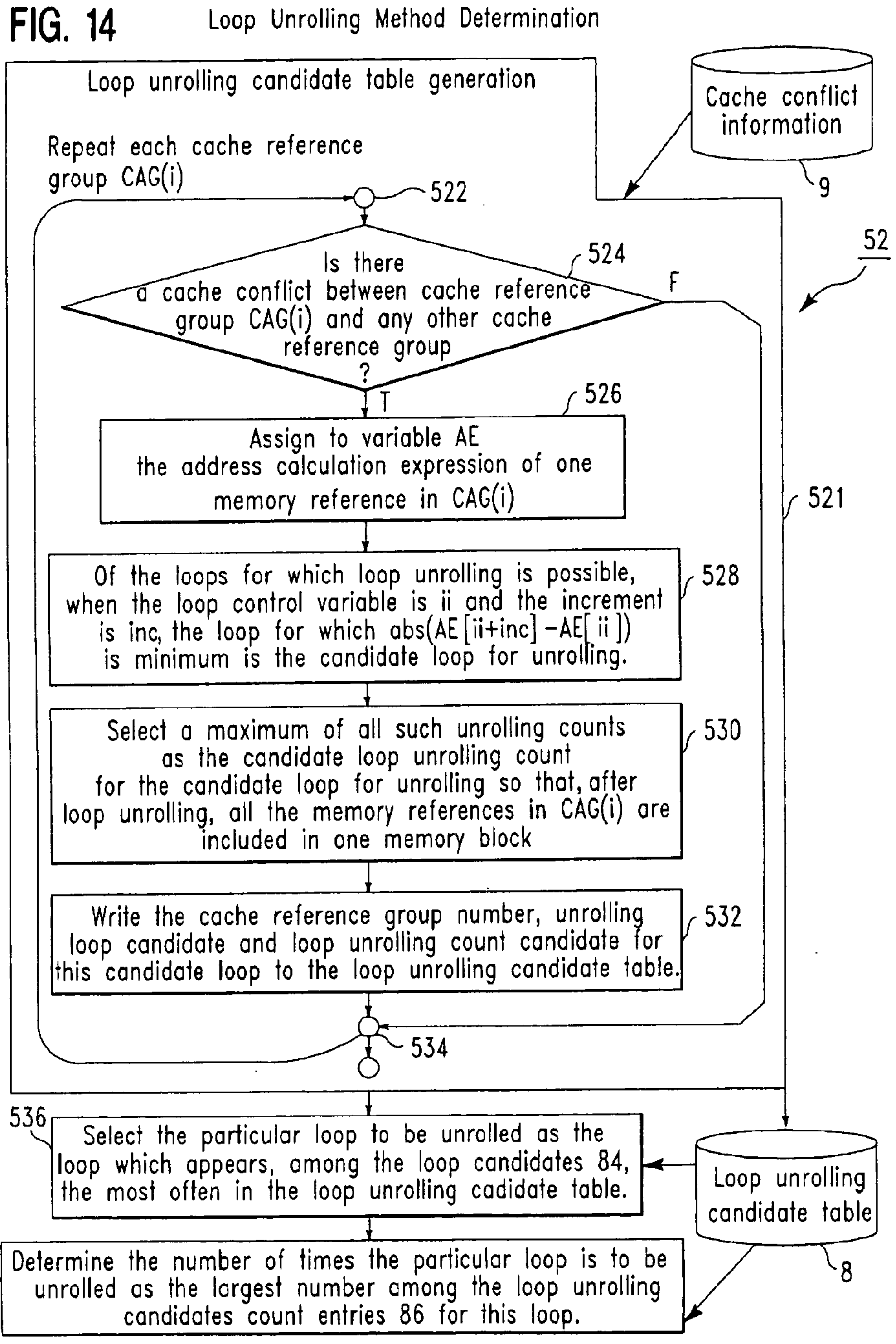

Loop Unrolling Method Determination
Loop unrolling candidate table generation
Repeat each cache reference group CAG(i)
522
Is there a cache conflict between cache reference group CAG(i) and any other cache reference group ?
524
F
T
526
Assign to variable AE the address calculation expression of one memory reference in CAG(i)
Of the loops for which loop unrolling is possible, when the loop control variable is ii and the increment is inc, the loop for which abs(AE[ii+inc] −AE[ii]) is minimum is the candidate loop for unrolling.
528
Select a maximum of all such unrolling counts as the candidate loop unrolling count for the candidate loop for unrolling so that, after loop unrolling, all the memory references in CAG(i) are included in one memory block
530
Write the cache reference group number, unrolling loop candidate and loop unrolling count candidate for this candidate loop to the loop unrolling candidate table.
532
534
521
Cache conflict information
9
52
536
Select the particular loop to be unrolled as the loop which appears, among the loop candidates 84, the most often in the loop unrolling cadidate table.
Loop unrolling candidate table
8
538
Determine the number of times the particular loop is to be unrolled as the largest number among the loop unrolling candidates count entries 86 for this loop.

FIG. 15

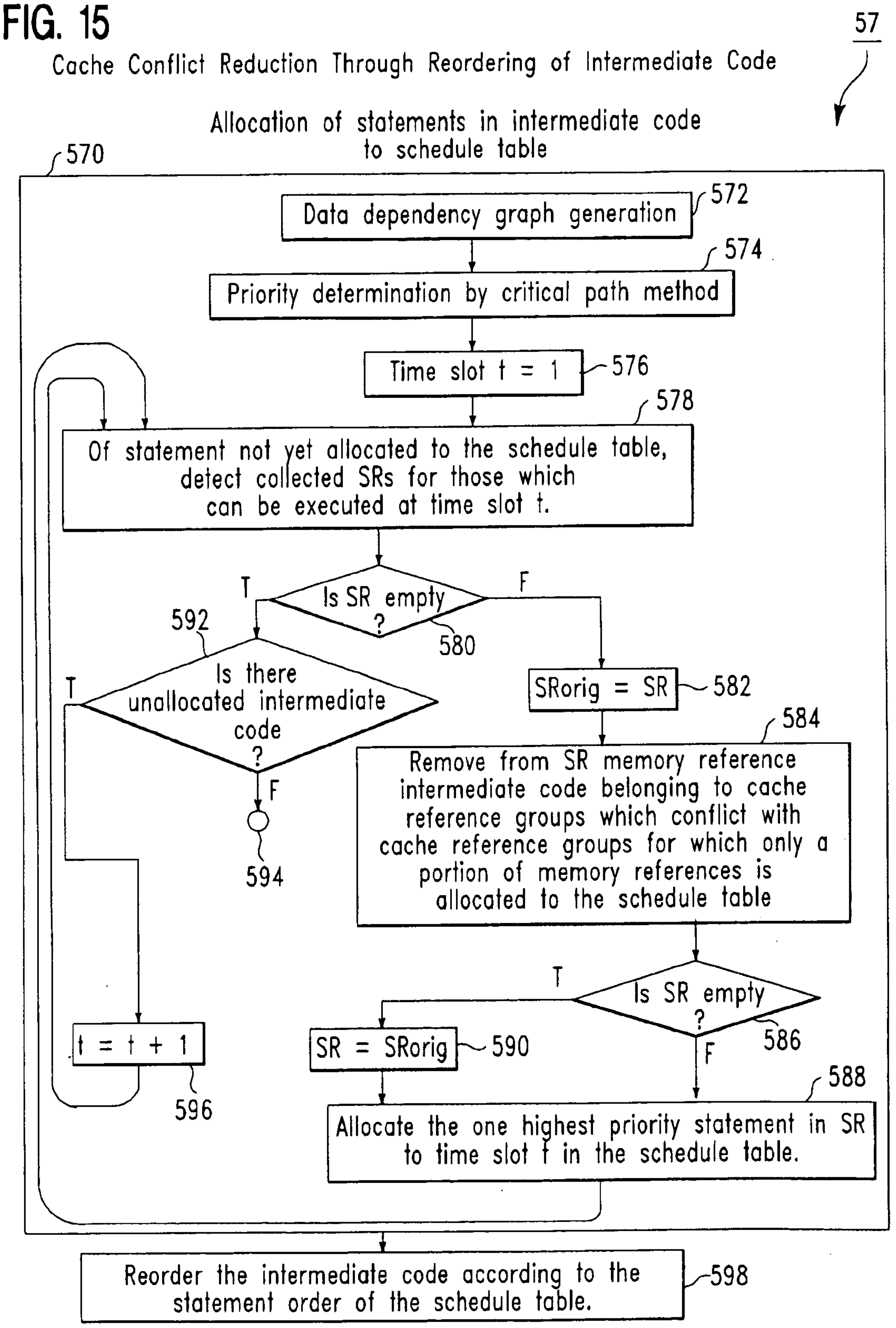

57
Cache Conflict Reduction Through Reordering of Intermediate Code
Allocation of statements in intermediate code to schedule table
570
Data dependency graph generation
572
Priority determination by critical path method
574
Time slot t = 1
576
Of statement not yet allocated to the schedule table, detect collected SRs for those which can be executed at time slot t.
578
Is SR empty ?
T
F
580
Is there unallocated intermediate code ?
592
T
F
594
SRorig = SR
582
Remove from SR memory reference intermediate code belonging to cache reference groups which conflict with cache reference groups for which only a portion of memory references is allocated to the schedule table
584
Is SR empty ?
T
F
586
t = t + 1
596
SR = SRorig
590
Allocate the one highest priority statement in SR to time slot t in the schedule table.
588
Reorder the intermediate code according to the statement order of the schedule table.
598

FIG. 16

5

Intermediate Code

305 Statement number	310 Label	320 Three address statement	330 Reference group number
1	loop:	T1 = A(I,J)	
2		T2 = B(I,J,2)	
3		T3 = T1 * T2	
4		T4 = A(I+1,J)	
5		T5 = B(I,J,3)	
6		T6 = T4 * T5	
7		T7 = T3 + T6	
8		T8 = B(I,J,1)	
9		T9 = T7 +T8	
10		C(I,J) = T9	
11		I = I + 1	
12		if I ≦ 255 goto loop	

FIG. 17

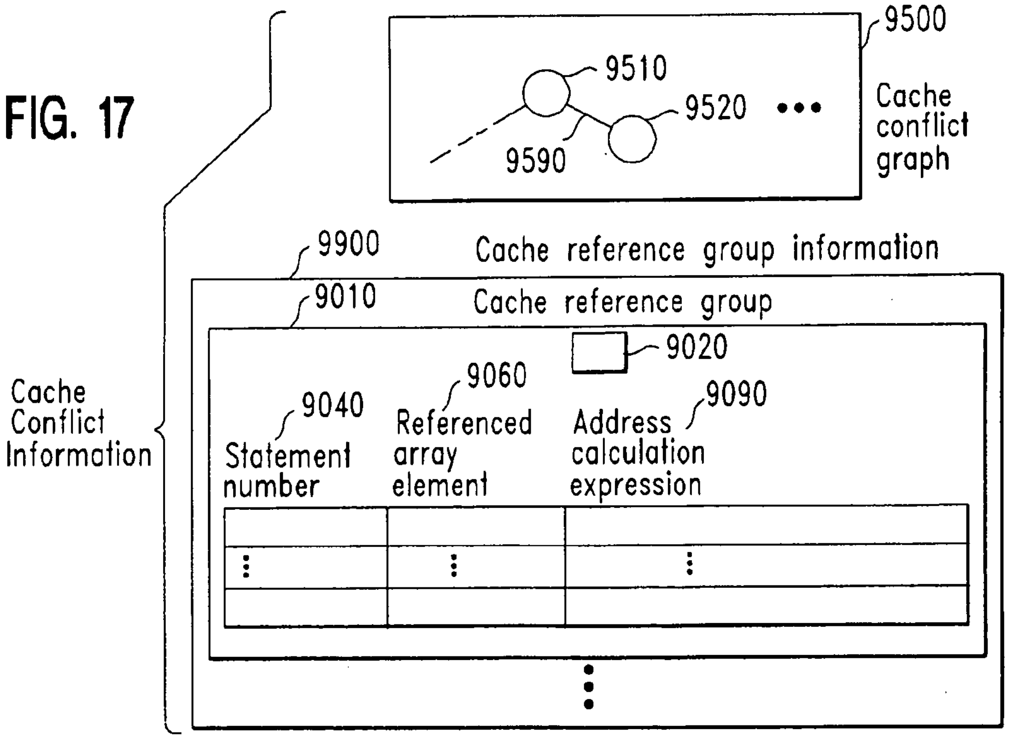
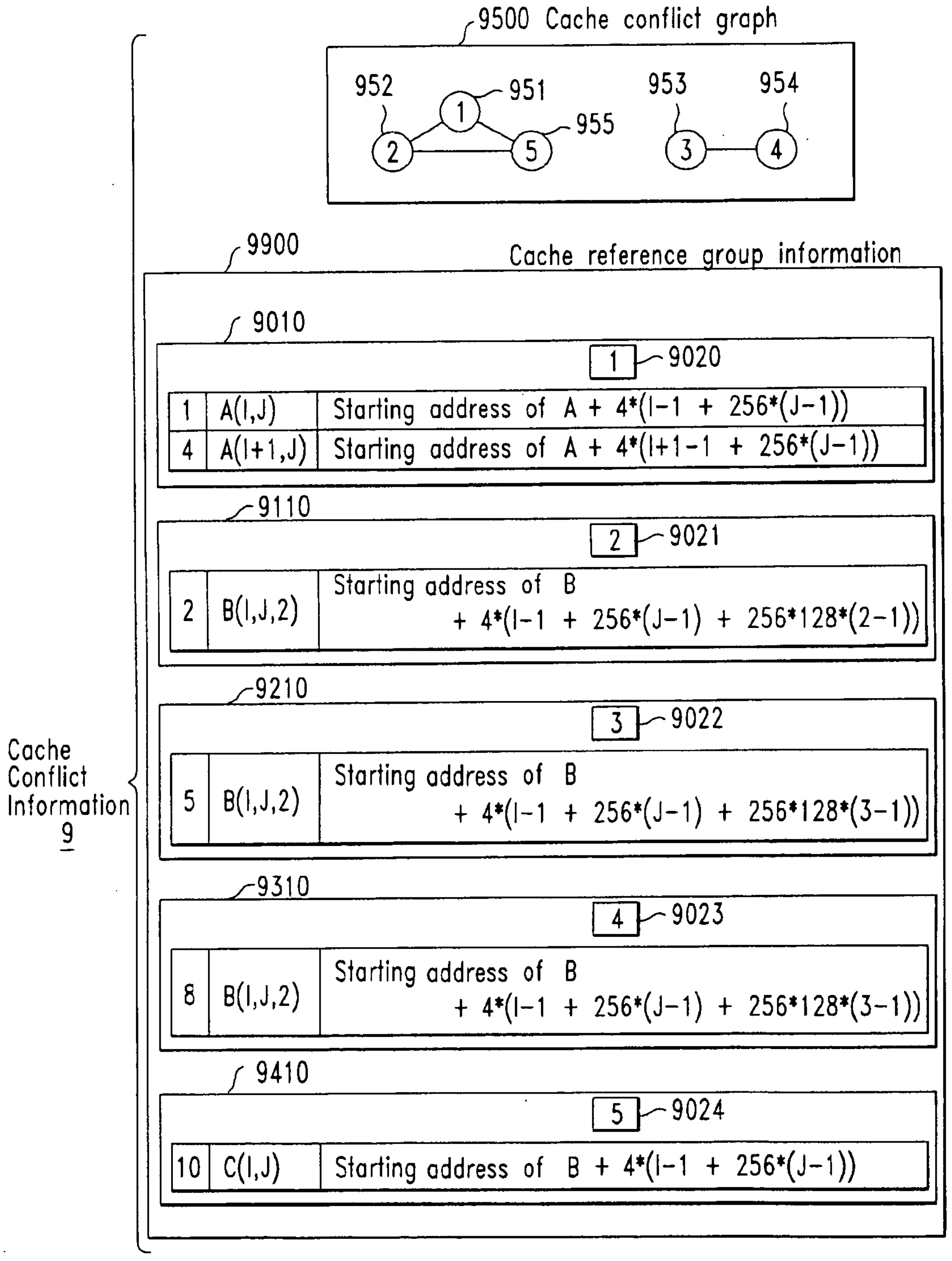


FIG. 18


9500 Cache conflict graph
952
951
955
953
954
1
2
5
3
4
9900
Cache reference group information
9010
9020
9110
9021
9210
9022
9310
9023
9410
9024
Cache Conflict Information 9

1	A(I,J)	Starting address of A + 4*(I-1 + 256*(J-1))
4	A(I+1,J)	Starting address of A + 4*(I+1-1 + 256*(J-1))

2	B(I,J,2)	Starting address of B + 4*(I-1 + 256*(J-1) + 256*128*(2-1))

5	B(I,J,2)	Starting address of B + 4*(I-1 + 256*(J-1) + 256*128*(3-1))

8	B(I,J,2)	Starting address of B + 4*(I-1 + 256*(J-1) + 256*128*(3-1))

10	C(I,J)	Starting address of B + 4*(I-1 + 256*(J-1))

FIG. 19

Intermediate Code

5

305	310	320	330
1	loop:	T1 = A(I,J)	1
2		T2 = B(I,J,2)	2
3		T3 = T1 * T2	
4		T4 = A(I+1,J)	1
5		T5 = B(I,J,3)	3
6		T6 = T4 * T5	
7		T7 = T3 + T6	
8		T8 = B(I,J,1)	4
9		T9 = T7 +T8	
10		C(I,J) = T9	5
11		I = I + 1	
12		if I ≦ 255 goto loop	

FIG. 22A

Loop Unrolling Candidate Table

8

82 Cache Reference Group Number	84 Unrolling Loop Candidate	86 Loop Unrolling Count Candidate
⋮	⋮	⋮

FIG. 22B

Loop Unrolling Candidate Table

8

82	84	86
1	DO 20 I=1,255	3
2	DO 20 I=1,255	4
3	DO 20 I=1,255	4
4	DO 20 I=1,255	4
5	DO 20 I=1,255	4

FIG. 20A

Intermediate Code

5

305	310	320	330
1	loop:	T1 = A(I,J)	1
2		T2 = B(I,J,2)	2
3		T3 = T1 * T2	
4		T4 = A(I+1,J)	1
5		T5 = B(I,J,3)	3
6		T6 = T4 * T5	
7		T7 = T3 + T6	
8		T8 = B(I,J,1)	4
9		T9 = T7 +T8	
10		C(I,J) = T9	5
11		T11 = A(I+1,J)	1
12		T12 = B(I+1,J,2)	2
13		T13 = T11 * T12	
14		T14 = A(I+2,J)	1
15		T15 = B(I+1,J,3)	3
16		T16 = T14 * T15	
17		T17 = T13 + T16	
18		T18 = B(I+1,J,1)	4
19		T19 = T17 + T18	
20		C(I+1,J) = T19	5
21		T21 = A(I+2,J)	1
22		T22 = B(I+2,J,2)	2
23		T23 = T21 * T22	
24		T24 = A(I+3,J)	1

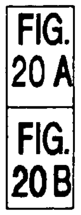


FIG. 20

FIG. 20B

Intermediate Code

5

305	310	320	330
25		T25 = B(I+2,J,3)	3
26		T26 = T24 * T25	
27		T27 = T23 + T26	
28		T28 = B(I+2,J,1)	4
29		T29 = T27 + T28	
30		C(I+2,J) = T29	5
31		T31 = A(I+3,J)	1
32		T32 = B(I+3,J,2)	2
33		T33 = T31 * T32	
34		T34 = A(I+4,J)	1
35		T35 = B(I+3,J,3)	3
36		T36 = T34 * T35	
37		T37 = T33 + T36	
38		T38 = B(I+3,J,1)	4
39		T39 = T37 + T38	
40		C(I+3,J) = T39	5
41		I = I + 4	
42		if I ≦ 252 goto loop	

FIG. 21A

FIG. 21A
FIG. 21B

FIG. 21

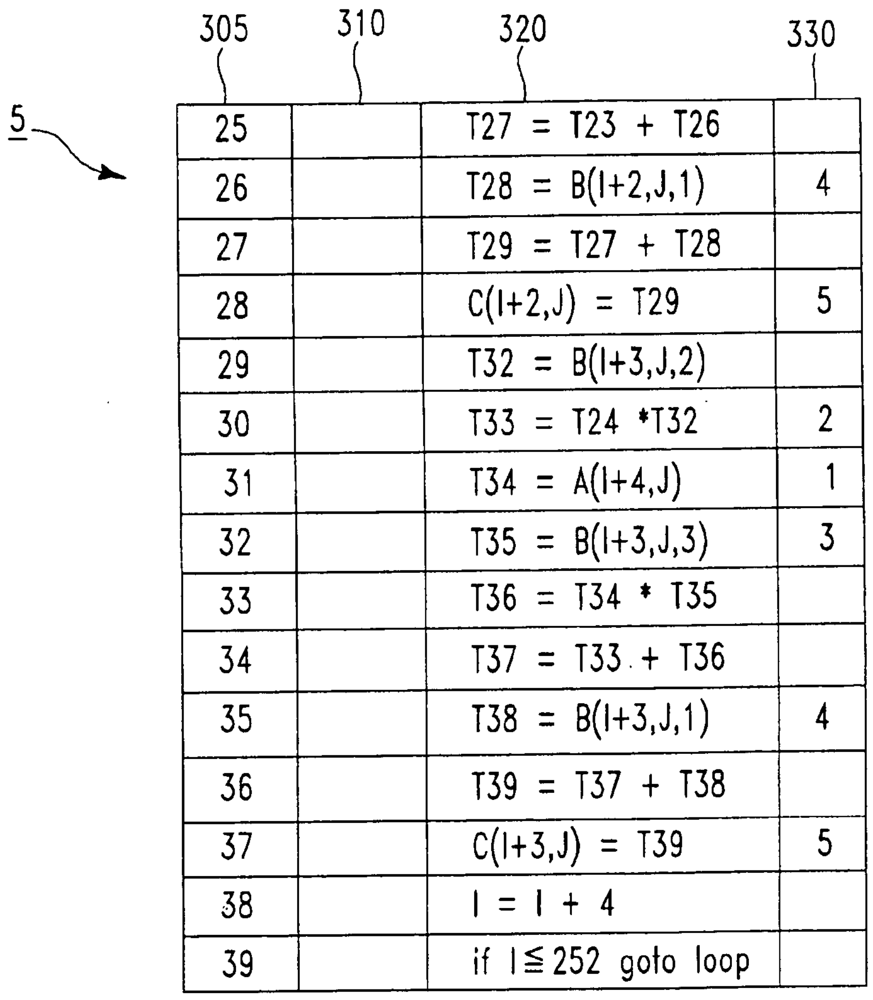
Intermediate Code

5

305	310	320	330
1	loop:	T1 = A(I,J)	1
2		T2 = B(I,J,2)	2
3		T3 = T1 * T2	
4		T4 = A(I+1,J)	1
5		T5 = B(I,J,3)	3
6		T6 = T4 * T5	
7		T7 = T3 + T6	
8		T8 = B(I,J,1)	4
9		T9 = T7 +T8	
10		C(I,J) = T9	5
11		T12 = B(I+1,J,2)	2
12		T13 = T4 * T12	
13		T14 = A(I+2,J)	1
14		T15 = B(I+1,J,3)	3
15		T16 = T14 * T15	
16		T17 = T13 + T16	
17		T18 = B(I+1,J,1)	4
18		T19 = T17 + T18	
19		C(I+1,J) = T19	5
20		T22 = B(I+2,J,2)	2
21		T23 = T14 * T22	
22		T24 = A(I+3,J)	1
23		T25 = B(I+2,J,3)	3
24		T26 = T24 * T25	

FIG. 21B

Intermediate Code


305
310
320
330
5

25		T27 = T23 + T26	
26		T28 = B(I+2,J,1)	4
27		T29 = T27 + T28	
28		C(I+2,J) = T29	5
29		T32 = B(I+3,J,2)	
30		T33 = T24 *T32	2
31		T34 = A(I+4,J)	1
32		T35 = B(I+3,J,3)	3
33		T36 = T34 * T35	
34		T37 = T33 + T36	
35		T38 = B(I+3,J,1)	4
36		T39 = T37 + T38	
37		C(I+3,J) = T39	5
38		I = I + 4	
39		if I≦252 goto loop	

FIG. 23

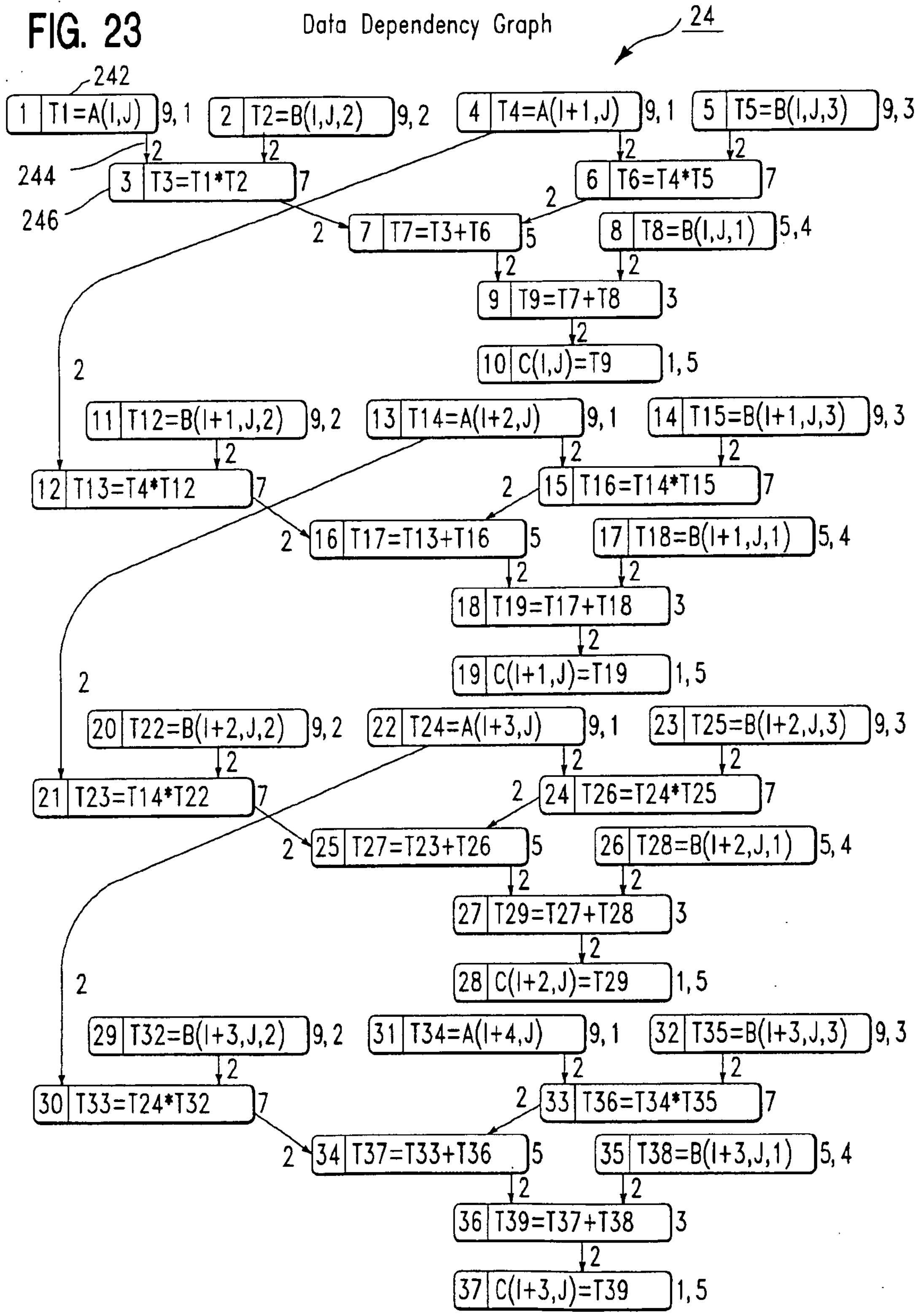

Data Dependency Graph
24
242
244
246
1 T1=A(I,J) 9,1
2 T2=B(I,J,2) 9,2
4 T4=A(I+1,J) 9,1
5 T5=B(I,J,3) 9,3
3 T3=T1*T2 7
6 T6=T4*T5 7
7 T7=T3+T6 5
8 T8=B(I,J,1) 5,4
9 T9=T7+T8 3
10 C(I,J)=T9 1,5
11 T12=B(I+1,J,2) 9,2
13 T14=A(I+2,J) 9,1
14 T15=B(I+1,J,3) 9,3
12 T13=T4*T12 7
15 T16=T14*T15 7
16 T17=T13+T16 5
17 T18=B(I+1,J,1) 5,4
18 T19=T17+T18 3
19 C(I+1,J)=T19 1,5
20 T22=B(I+2,J,2) 9,2
22 T24=A(I+3,J) 9,1
23 T25=B(I+2,J,3) 9,3
21 T23=T14*T22 7
24 T26=T24*T25 7
25 T27=T23+T26 5
26 T28=B(I+2,J,1) 5,4
27 T29=T27+T28 3
28 C(I+2,J)=T29 1,5
29 T32=B(I+3,J,2) 9,2
31 T34=A(I+4,J) 9,1
32 T35=B(I+3,J,3) 9,3
30 T33=T24*T32 7
33 T36=T34*T35 7
34 T37=T33+T36 5
35 T38=B(I+3,J,1) 5,4
36 T39=T37+T38 3
37 C(I+3,J)=T39 1,5
2

FIG. 24A

Schedule Table

26

303 Time Slot Value	305 Statement Number	320 Three Address Statement	330 Reference Group Number
1	1	T1 = A(I,J)	1
2	4	T4 = A(I+1,J)	1
3	5	T5 = B(I,J,3)	3
4	13	T14 = A(I+2,J)	1
5	14	T15 = B(I+1,J,3)	3
6	22	T24 = A(I+3,J)	1
7	23	T25 = B(I+2,J,3)	3
8	31	T34 = A(I+4,J)	1
9	32	T35 = B(I+3,J,3)	3
10	2	T2 = B(I,J,2)	2
11	11	T12 = B(I+1,J,2)	2
12	20	T22 = B(I+2,J,2)	2
13	29	T32 = B(I+3,J,2)	2
14	3	T3 = T1 * T2	
15	6	T6 = T4 * T5	
16	12	T13 = T4 * T12	
17	15	T16 = T14 * T15	
18	21	T23 = T14 * T22	
19	24	T26 = T24 * T25	
20	30	T33 = T24 * T32	
21	33	T36 = T34 * T35	
22	7	T7 = T3 + T6	
23	8	T8 = B(I,J,1)	4
24	16	T17 = T13 + T16	

FIG. 24A
FIG. 24B

FIG. 24

FIG. 24B

Schedule Table

26

303	305	320	330
25	17	T18 = B(I+1,J,1)	4
26	25	T27 = T23 +T26	
27	26	T28 = B(I+2,J,1)	4
28	34	T37 = T33 + T36	
29	35	T38 = B(I+3,J,1)	4
30	9	T9 = T7 + T8	
31	18	T19 = T17 + T18	
32	27	T29 = T27 + T28	
33	36	T39 = T37 + T38	
34	10	C(I,J) = T9	5
35	19	C(I+1,J) = T19	5
36	28	C(I+2,J) = T29	5
37	37	C(I+3,J) = T39	5
38	38	I = I + 4	
39	39	if I ≦ 252 goto loop	

FIG. 25A

FIG. 25B

FIG. 25

FIG. 25A

Intermediate Code

5

303	310	320	330
1	loop:	T1 = A(I,J)	1
2		T4 = A(I+1,J)	1
3		T5 = B(I,J,3)	3
4		T14 = A(I+2,J)	1
5		T15 = B(I+1,J,3)	3
6		T24 = A(I+3,J)	1
7		T25 = B(I+2,J,3)	3
8		T34 = A(I+4,J)	1
9		T35 = B(I+3,J,3)	3
10		T2 = B(I,J,2)	2
11		T12 = B(I+1,J,2)	2
12		T22 = B(I+2,J,2)	2
13		T32 = B(I+3,J,2)	2
14		T3 = T1 * T2	
15		T6 = T4 * T5	
16		T13 = T4 * T12	
17		T16 = T14 * T15	
18		T23 = T14 * T22	
19		T26 = T24 * T25	
20		T33 = T24 * T32	
21		T36 = T34 * T35	
22		T7 = T3 + T6	
23		T8 = B(I,J,1)	4
24		T17 = T13 + T16	

FIG. 25B

Intermediate Code

5

303	310	320	330
25		T18 = B(I+1,J,1)	4
26		T27 = T23 +T26	
27		T28 = B(I+2,J,1)	4
28		T37 = T33 + T36	
29		T38 = B(I+3,J,1)	4
30		T9 = T7 + T8	
31		T19 = T17 + T18	
32		T29 = T27 + T28	
33		T39 = T37 + T38	
34		C(I,J) = T9	5
35		C(I+1,J) = T19	5
36		C(I+2,J) = T29	5
37		C(I+3,J) = T39	5
38		I = I + 4	
39		if I ≦ 252 goto loop	

FIG. 26A

Object Program

7

61 Instruction Number	62 Label	63 Instruction	64 Operand
1	loop:	fload	A(r1,J),fr1
2		fload	A(r1+1,J)fr2
3		fload	B(r1,J,3),fr3
4		fload	A(r1+2,J),fr4
5		fload	B(r1+1,J,3),fr5
6		fload	A(r1+3,J),fr6
7		fload	B(r1+2,J,3),fr7
8		fload	A(r1+4,J),fr8
9		fload	B(r1+3,J,3),fr9
10		fload	B(r1,J,2),fr10
11		fload	B(r1+1,J,2),fr11
12		fload	B(r1+2,J,2),fr12
13		fload	B(r1+3,J,2),fr13
14		fmpy	fr1,fr10,fr10
15		fmpy	fr2, fr3,fr3
16		fmpy	fr2,fr11,fr11
17		fmpy	fr4, fr5,fr5
18		fmpy	fr4, fr12,fr12
19		fmpy	fr6,fr7,fr7
20		fmpy	fr6,fr13,fr13
21		fmpy	fr8,fr9,fr9
22		fadd	fr10,fr3,fr3
23		fload	B(r1,J,1),fr14
24		fadd	fr11,fr5,fr5

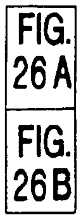


FIG. 26

FIG. 26B

Object Program

7

Instruction Number (61)	Label (62)	Instruction (63)	Operand (64)
25		fload	B(r1+1,J,1),fr15
26		fadd	fr12,fr7,fr7
27		fload	B(r1+2,J,1),fr16
28		fadd	fr13,fr9,fr9
29		fload	B(r1+3,J,1),fr17
30		fadd	fr3,fr14,fr14
31		fadd	fr5,fr15,fr15
32		fadd	fr7,fr16,fr16
33		fadd	fr9,fr17,fr17
34		fstore	fr14,C(r1,J)
35		fstore	fr15,C(r1+1,J)
36		fstore	fr16,C(r1+2,J)
37		fstore	fr17,C(r1+3,J)
38		add	r1,#4,r1
39		cmp	r1,#252
40		ble	loop

FIG. 27

70	71	72	73	74	75	76
I Number	Instruction number	Referenced array element	Hit status	Accepted block	Rejected block	Cache miss caused by cache conflict
	1	A(1,J)	Miss	A(1:4,J)		
	2	A(2,J)	Hit			
	3	B(1,J,3)	Miss	B(1:4,J,3)		
	4	A(3,J)	Hit			
	5	B(2,J,3)	Hit			
	6	A(4,J)	Hit			
	7	B(3,J,3)	Hit			
	8	A(5,J)	Miss	A(5:8,J)		
	9	B(4,J,3)	Hit			
I = 1	10	B(1,J,2)	Miss	B(1:4,J,2)	A(1:4,J)	
	11	B(2,J,2)	Hit			
I = 4	12	B(3,J,2)	Hit			
	13	B(4,J,2)	Hit			
	23	B(1,J,1)	Miss	B(1:4,J,1)	B(1:4,J,3)	
	25	B(2,J,1)	Hit			
	27	B(3,J,1)	Hit			
	29	B(4,J,1)	Hit			
	34	C(1,J)	Miss	C(1:4,J)	B(1:4,J,2)	
	35	C(2,J)	Hit			
	36	C(3,J)	Hit			
	37	C(4,J)	Hit			

⋮

FIG. 28

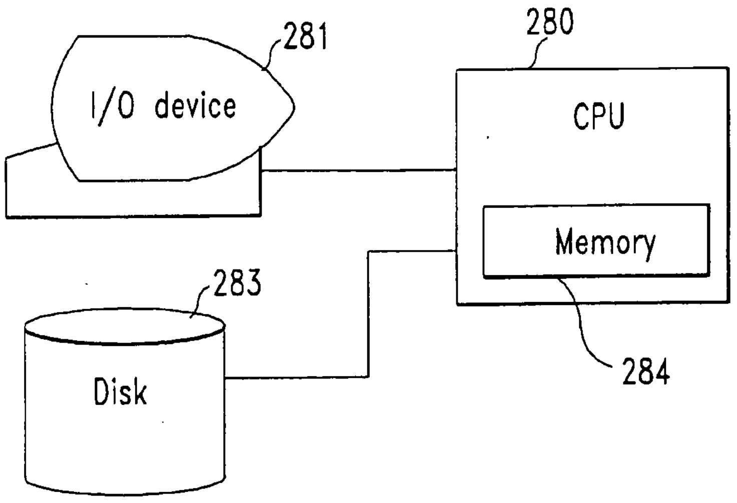
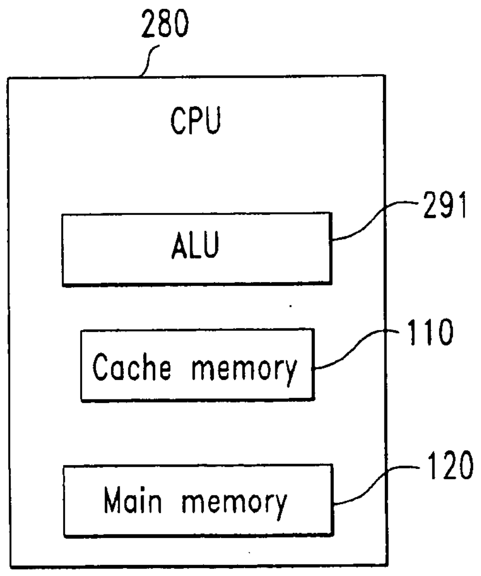

281
I/O device
280
CPU
Memory
283
Disk
284

FIG. 29


280
CPU
ALU
291
Cache memory
110
Main memory
120

COMPILE METHOD FOR REDUCING CACHE CONFLICT

CROSS REFERENCE TO RELATED APPLICATION

This application is a continuation of copending patent application Ser. No. 08/303,007, filed Sep. 8, 1994, now abandoned entitled "Compile Method for Reducing Cache Conflict."

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention concerns a method of compiling programs which are executed on computers having cache memory, and more specifically, such a method which generates code that reduces, by as much as possible, cache conflicts which would otherwise occur due to conflicting cache access during execution of these programs.

2. Description of the Prior Art

Cache memory and cache conflict are described in works such as J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* (©1990 Morgan Kaufmann Publishers, Inc., Palo Alto, Calif.), pages 408–425.

Cache memory (referred to hereafter simply as "cache") is a type of rapidly accessible memory which is used between a processing device and main memory. By placing a copy of one portion of data residing in main memory into cache, data referencing speed, i.e., the speed at which this data is accessed, can be increased. The units of data transferred between cache and main memory are called "blocks"; blocks in cache are called "cache blocks", while blocks in main memory are called "memory blocks".

As to methods of implementing caches, there are three mapping methods which place copies of memory blocks into cache blocks: the full associative method, the set associative method, and the direct map method. Recently, to provide caches with enhanced capacity and increased speed, the direct map method is coming into mainstream use.

With the direct map method, the cache blocks, into which memory blocks are mapped, are uniquely determined. Since cache capacity is generally smaller than main memory capacity, multiple memory blocks are mapped into one cache block. Because of this, even if a piece of data has been transferred to a cache block, if the data that was previously written into that cache block is now rejected from the cache block due to a reference of another memory block now mapped in the same cache block, a cache miss will be generated with the next reference.

This phenomenon is called "cache conflict"; cache misses generated by this phenomenon are called "cache misses caused by cache conflict". One drawback of the direct map method is that with certain programs, a substantial amount of cache conflict is generated, causing a marked drop in performance.

FIGS. 2 through 7 show cache conflict in detail. Hereafter, and for purposes of illustration and simplicity, I will assume the use of the direct map method with a block length of 16 bytes and a cache capacity of 256Kbytes (256x1024 bytes).

FIG. 2 shows a source program; FIG. 3 shows an overview of the status of mapping from main memory to cache; FIGS. 4 and 5 show, in increased detail over that in FIG. 3, the status of mapping from main memory to cache; FIG. 6 shows an example of an object program for the source program of FIG. 2; and FIG. 7 shows the generation of a cache conflict.

First, FIG. 2 shows an illustration program, here a source program, in which multiple cache conflicts are generated. In this source program 3, with a common declaration, arrays A, B, and C are allocated consecutively in the memory area in this order. Arrays A and C are two dimensional arrays; array B is a three dimensional array. Each array is a real type (floating point numbers) with element length of 4 bytes. The declared size for array A is (256, 128), for array B is (256, 128, 3), and array C is (256, 128). The execution portion of program 3 is a doubly nested loop. For the innermost loop, for I=1 to 255, the C(I,J) value is calculated using the values A(I,J), B(I,J,2), A(I+1,J), B(I,J,3), and B(I,J,1).

FIG. 3 shows a part of each array residing within main memory 120 and its corresponding part of cache 110, and the correspondence relationship itself. The size of each area in the memory that contains the parts of arrays, specifically A(1:256,1:128), B(1:256,1:128,1), B(1:256,1:128,2), B(1:256,1:128,3), and C(1:256,1:128), is 128Kbytes (=256*128*4). Accordingly, arrays A(1:256,1:128), B(1:256,1:128,2), and C(1:256,1:128) are mapped in one common area in cache 110, while arrays B(1:256,1:128,1) and B(1:256,1:128,3) are also mapped in another common area in cache 110. The above mentioned expressions "m:n" show the range of subscripts from lower bound m to upper bound n.

FIGS. 4 and 5 collectively show, in detail, the structure of cache 110 as well as its mapping with main memory 120.

As shown, cache 110 is a collection of unitized 16 byte cache blocks, illustratively cache blocks 111 to 116. Main memory 120 is also shown with unitized memory blocks. Mapping from memory block 120 to cache 110 is shown with arrows.

For example, in FIG. 4, three memory blocks A(1:4,J), B(1:4,J,2), and C(1:4,J) are mapped into one common cache block 111. However, only one of these three memory blocks can reside in cache block 111 at any one time. Because of this, cache conflict can arise where the data is referenced in these memory blocks through the cache. Similarly, cache conflict also occurs in cache blocks 112 and 113.

Since, as shown in FIG. 5, two memory blocks belonging to the same array (B) are mapped into each one of three common cache blocks, cache conflict can be generated here as well. Consequently, as shown in this figure, cache conflict is also generated for accesses within the same array (B).

FIG. 6 is an example of an object program that corresponds to source program 3 shown in FIG. 2. However, FIG. 6 only shows the object code for the innermost loop in FIG. 2. Since generally during execution of the program, the greater portion of the processing time is spent executing the innermost loop, the remaining portions, at least for this discussion, are not very important and, for simplicity, will be ignored.

In FIG. 6, each instruction is identified by a unique instruction number 61 (specifically numerals 1—13) added to the left side of the figure. The label 62 is used as a branch target for branch instructions. Processing contents 65 of each instruction 63 for operand 64 are shown all the way to the right in this figure. Of these instructions, instructions 1, 2, 4, 5, 8 and 10 are memory reference instructions, inasmuch as they reference array elements A(I,J), B(I,J,2), A(I+1,J), B(I,J,3), B(I,J,1), and C(I,J) respectively.

In FIG. 7, the cache conflicts caused by memory reference instructions in the object program, shown in FIG. 6, are depicted in the order in which these instructions are executed. Each line shows, from the left: a value 70 (I value) for innermost loop control variable I; instruction number 71 with each different instruction carrying its specific instruction number from FIG. 6; instruction referenced array element **72**; cache hit status **73** (shows whether or not referenced data exists in cache); accepted block **74**, i.e., the memory block which was transferred from main memory **120** and placed in cache **110**; rejected block **75**, i.e., the memory block which was rejected from cache **110**, and cache miss **76** which shows whether this miss was caused by a cache conflict.

As can be seen from FIG. **7**, cache misses occur with all of the memory references, with most of these cache misses caused by cache conflict. For example, instructions **1** and **4** for I=1 both reference the same memory block A(1:4,J), but because memory block B(1,J,2) which is now mapped in the same cache block as was block A(1:4,J) previously referenced by instruction **2**, block A(1:4,J) is rejected from cache. This causes a cache miss to occur with instruction **4**.

Of the cache misses shown in FIG. **7**, the cache misses generated by referencing the same memory block twice are all cache misses caused by cache conflict. Of the 24 cache misses from I=1 to I=4, 18 of them are cache misses caused by cache conflict. Thus, it is clear that cache conflict is a major reason for cache misses.

One method of reducing cache conflict is using the full associative method or the set associative method as shown in the above prior art. However, with either of these methods, the implementing caching hardware becomes complex and cache referencing speed is reduced. Furthermore, with each of these methods, it is difficult to significantly increase the capacity of the cache.

Moreover, with set associative method based cache, when the number of cache block candidates (associativity) to which the memory blocks are mapped is small, it is not possible to sufficiently avoid cache conflict. Consequently, for this type of cache, even when the associativity is small, use of a cache conflict reduction method is still quite necessary.

With the direct map method, cache conflict may be reduced by simply increasing the capacity of cache memory. However, when compared with the cost of main memory of the same capacity, cache memory costs significantly more than main memory, hence limits exist to increasing cache capacity.

As shown above, because the structure of direct map method cache is simple, this method has the advantages of providing enhanced access speed and ease of expanding cache capacity. However, with certain programs, disadvantageously cache conflict occurs and performance is greatly lessened. Until the present invention, as described below, a sufficient way did not exist to reduce cache conflict in direct map method cache.

Even with set associative method based cache which generates relatively little cache conflict, nevertheless when associativity is small, cache conflict occurs and performance is greatly lessened. If the associativity is sufficiently increased, all of the cache conflict can be avoided, but at the cost of complicating the ensuing caching hardware structure, lowering cache referencing speed and making any significant increase in cache capacity quite difficult. Based on these reasons, the associativity of the set associative method based cache currently available on the commercial market is approximately 2 to 4, which, for programs that might generate an appreciable amount of cache conflict, is inadequate, in practice, to sufficiently prevent the cache conflict from occurring.

SUMMARY OF THE INVENTION

The purpose of this invention is to provide a method for reducing cache conflict for programs that are otherwise susceptible to a marked decrease in performance due to cache conflict.

To achieve this purpose in computers with cache memory, the present invention detects, during program compilation into object code, the generation of cache conflict, which occurs between memory references in the input program. The invention also performs cache conflict analysis to determine which specific memory references cause the cache conflict.

Based on the results of this analysis, when a cache conflict is detected and before all references would have been completed to a memory block previously transferred to a cache block, i.e., before all those references would be embodied in compiled object code ready to be executed, the memory references in the compiled object program are reordered such that the memory block would now not be rejected from cache.

The reordering of memory references is done on the compiler intermediate code or on the object program generated by the compiler. Alternatively, memory references on the compiler intermediate code can be reordered followed by a further reordering of the appropriate references on the object program generated by the compiler. Furthermore, the memory references on the program intermediate code can be reordered with resulting reordered intermediate code then being reverse converted (reverse compiled) back to a new source program version. This new version should ultimately generate far fewer, if any, cache conflicts. That version is then provided as an output source program for subsequent compilation and execution either in the computer which generated this program or another computer.

My inventive method for use with computers with cache memories includes a cache conflict analysis step for detecting cache conflicts related to memory references in input source programs that are being compiled into object code. This step provides, as output cache conflict information, analyzed cache conflict generation conditions. In conjunction with the analysis step, my inventive method includes a memory reference reordering step that changes the order of memory references in the source program, as it is being compiled, and particularly in corresponding intermediate code therefor, so that a memory block is not rejected from cache before all references to memory blocks, previously transferred to a common cache block, would have been completed.

The above cache conflict analysis step includes a cache reference group analysis step in which memory references in the input program are extracted, and, thereafter, memory references to the same memory block are grouped as a cache reference group. During the latter step, the cache reference groups are classified with cache reference group information being then generated as cache conflict information. After the cache conflict information is generated, a cache conflict graph analysis step is performed through which, based on the cache reference group information, a graphical output depiction of cache conflict information is provided which shows the cache conflict conditions among different cache reference groups.

When address distances, in main memory, among the memory references in the source program are below a designated value, the cache reference group analysis step concludes that these memory references are referencing the same memory block, and assigns these references to the same cache reference group.

The cache reference group analysis step obtains memory blocks referenced to positions (addresses) in main memory of the memory references in the input program, and, for those memory references which refer to a common memory block, assigns those references to the same cache reference group.

For arbitrary first and second cache reference groups included in the cache reference group information, the above cache conflict graph analysis step obtains a minimum value cache distance between all of the memory references included in the first cache reference group and all of the memory references included in the second cache reference group. When the minimum value is below the designated value, cache conflict is viewed as being generated between the memory references of the first cache reference group and the memory references of the second cache reference group.

Furthermore, for the arbitrary first and second cache reference groups, the cache conflict graph analysis step also obtains the cache blocks mapped from the main memory position of all the memory references included in the first cache reference group, and likewise obtains the cache blocks mapped from the main memory position of all the memory references included in the second cache reference group. When there are memory references mapped to the same cache block, this step concludes that cache conflict is generated between the memory references of the first cache reference group and the memory references of the second cache reference group.

After performing loop unrolling for a loop in the input program, the memory references reordering step, described above, changes the order of instructions to coincide with the order of the memory references. In particular, the loop that is to be unrolled and the loop unrolling count therefor are both selected such that the loop has a memory reference range length of approximately the same length, once the loop is unrolled, as the cache blocks.

As a result of using my inventive method with each of those programs which would otherwise exhibit cache conflicts and hence lowered performance, a corresponding object program or associated source program is generated that manifests fewer, if any, cache conflicts.

By detecting when cache conflicts would be generated in the input program, then, in accordance with my present invention, specific processing can then be performed on that program which mainly focuses on eliminating these cache conflicts. Generally, when a cache conflict occurs, a marked decrease in performance results. Therefore, cache conflict reduction should be made a priority, for specified items, through detection of cache conflict generation areas.

By doing cache conflict analysis, with the concomitant reordering of memory references, those memory references which would otherwise cause each cache conflict can be determined.

Moreover, by reordering these memory references, cache conflict, in those memory areas where cache conflict would otherwise occur, can be avoided. For example, in order for a memory block not to be rejected from cache before all references, in that input program, to memory blocks previously transferred to a cache block would have been completed, cache misses caused by cache conflict can be completely and advantageously avoided, as I now teach, by changing the order of the memory references in the program.

By reordering the memory references for the intermediate code, redundant data dependency, heretofore caused by allocating a small number of registers to memory reference instructions, is eliminated. This, in turn, advantageously increases the degree of freedom for changing the order of memory references. Because of this, memory references can be moved to optimal positions with a concomitant improvement in the amount of cache conflict reduction that results.

Advantageously, by reordering the memory references for the object program, all instructions including those which

could not be foreseen at the intermediate code stage could be reordered. Hence, a very accurate instruction reordering could be accomplished to further reduce cache conflicts.

Furthermore, by reordering the memory references on both the program intermediate code and the object program, an optimal reduction in cache conflict could be obtained.

Additionally, by reordering the memory references for the program intermediate code, then reverse compiling the results to a source program with this latter program being supplied as an output source program for later use, any subsequent compiler operating on the output source program will, a priori, then generate code having reduced cache conflicts. Thus, through use of my inventive method, on one computer to generate source code with reduced cache conflicts and intended for eventual widespread distribution and execution on other computers, cache conflict reduction can be employed on a widespread basis thus increasing and possibly maximizing its general use.

By unrolling loops for reordering memory references, cache conflicts for multiple loop iterations can be eliminated, thereby further improving cache conflict reduction. In particular, by appropriately selecting the particular loops to be unrolled and the number of times each such loop is to be unrolled, the fewest loops will be unrolled consistent with highly effective loop unrolling. This, in turn, advantageously avoids having insufficient registers, and its attendant difficulties, which would otherwise occur whenever an excessive number of loops is to be unrolled.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of an example of a compiler using the compiling method of this invention;

FIG. **2** shows an example of conventional source program **3** that can generate cache conflicts;

FIG. **3** shows an overview of conventional main memory mapping to cache for illustrative program **3** depicted in FIG. **2**;

FIGS. **4** and **5** collectively show, in detail, the conventional main memory mapping to cache depicted in simplified form in FIG. **3**;

FIG. **6** shows a corresponding conventional object program for an inner-loop portion of source program **3** depicted in FIG. **2**;

FIG. **7** shows conventional cache conflict generation that would occur for the object program shown in FIG. **6**;

FIG. **8** is a high-level flowchart of cache conflict analysis process **30** shown in FIG. **1**;

FIG. **9** shows the proper orientation of the drawing sheets for FIGS. **9A** and **9B**;

FIGS. **9A** and **9B** collectively show a detailed flowchart of cache reference group analysis step **32** shown in FIG. **8**;

FIG. **10** is a detailed flowchart of distance calculation step **3218** executed within cache reference group analysis step **32** shown in FIGS. **9A** and **9B**;

FIG. **11** is a detailed flowchart of cache conflict graph analysis step **34** shown in FIG. **8**;

FIG. **12** is a detailed flowchart of distance calculation step **3408** executed within cache conflict graph analysis step **34** shown in FIG. **11**;

FIG. **13** is a high-level, summarized, flowchart of memory reference reordering process **50** shown in FIG. **1**;

FIG. **14** is a detailed flowchart of loop unrolling method determination step **52** shown in FIG. **13**;

FIG. **15** is a detailed flowchart of cache conflict elimination by intermediate code reordering process **57** shown in FIG. **13**;

FIG. 16 shows illustrative intermediate code, for a portion of source program 3 shown in FIG. 2, and prior to processing that intermediate code to reduce cache conflicts therein;

FIG. 17 shows the general structure of cache conflict information, e.g., cache conflict information 9 shown in FIG. 1;

FIG. 18 shows cache conflict information that results from use of my inventive method for exemplary intermediate code 5 for source program 3 shown in FIGS. 16 and 2, respectively;

FIG. 19 shows the illustrative intermediate code shown in FIG. 16 but after the cache reference group numbers for cache conflicts have been written into this code;

FIG. 20 shows the proper orientation of the drawing sheets for FIGS. 20A and 20B;

FIGS. 20A and 20B collectively show the illustrative intermediate code shown in FIG. 19 but after both loop unrolling and variable renaming have been completed therein through execution of steps 53 and 54, respectively, shown in FIG. 13;

FIG. 21 shows the proper orientation of the drawing sheets for FIGS. 21A and 21B;

FIGS. 21A and 21B collectively show intermediate code in FIGS. 20A and 20B but after common array elements have been eliminated from this code through execution of step 55 shown in FIG. 13;

FIGS. 22A and 22B respectively show the structure of a loop unrolling candidate table in general, and, in particular, a loop unrolling candidate table for illustrative intermediate code 5 as it undergoes loop unrolling;

FIG. 23 shows a data dependency graph for intermediate code 5 shown in FIGS. 21A and 21B;

FIG. 24 shows the proper orientation of the drawing sheets for FIGS. 24A and 24B;

FIGS. 24A and 24B collectively show an illustrative schedule table to which statements in intermediate code 5 shown in FIGS. 21A and 21B have been allocated according to data dependency graph 24 shown in FIG. 23;

FIG. 25 shows the proper orientation of the drawing sheets for FIGS. 25A and 25B;

FIGS. 25A and 25B collectively show intermediate code 5 but after complete execution of my inventive cache conflict reduction method on the intermediate code shown in FIGS. 21A and 21B;

FIG. 26 shows the proper orientation of the drawing sheets for FIGS. 26A and 26B;

FIGS. 26A and 26B collectively show a resulting object program after execution of my inventive cache conflict reduction method and specifically that which results after object level compilation has been performed on intermediate code 5 shown in FIGS. 25A and 25B;

FIG. 27 shows the cache conflict generation that occurs for object code 7 shown in FIGS. 26A and 26B;

FIG. 28 shows the structure of a computer system for implementing my present invention; and

FIG. 29 shows the structure of the central processing unit (CPU) in the computer system, depicted in FIG. 28 and used to execute a program generated by the compile method of my present invention.

To facilitate reader understanding, identical reference numerals are used to denote identical or similar elements that are common to the figures.

DESCRIPTION OF THE PREFERRED EMBODIMENT

I will now describe the preferred embodiment of the invention while referring to the figures, several of which may be simultaneously referred to during the course of the following description.

FIG. 1 shows the structure of an example of a compiler using the compile method of the present invention. FIG. 28 shows the structure of a computer system for implementing this invention. As shown in FIGS. 1 and 28, compiler 1 resides in memory 284 of central processing unit (CPU) 280, the program (and accompanying data) to be compiled resides in disk 283, and the display of analysis results and user instructions is presented through input/output (I/O) device 281. FIG. 29 shows the structure of CPU 280 which is executed by a program compiled by my inventive method. Within CPU 280, cache memory 110 (hereafter referred to simply as "cache") is installed between main memory 120 and arithmetic and logic unit (ALU) 291, the latter executing the program stored in main memory 120. The CPU also holds a portion of the program and accompanying data, all of which stored in main memory 120.

Now specifically referring to FIG. 1 and to facilitate understanding, I will now summarize the processing undertaken by this preferred embodiment as it relates to my invention.

Compiler 1 accepts source program 3, as input, and ultimately generates object program 7. Compiler 1 performs the following processes in order: syntax analysis process 10, cache conflict reduction process 20, and code generation process 60. With widely known technology, syntax analysis process 10 converts source program 3 into intermediate code 5 for subsequent use by the compiler. Cache conflict reduction process 20 is performed on intermediate code 5. Code generation process 60 generates object program 7 from intermediate code 5 but as altered by cache conflict reduction process 20.

Consequently, as readily seen, object program 7 is generated for input source program 3 but upon which cache conflict reduction process 20 was performed.

Next, I will explain the specific processing undertaken by cache conflict reduction process 20 which is characteristic of my invention. This processing contains three basic steps, with the processing composed of these three steps used on the intermediate code of the loops included within intermediate code 5. For purposes of simplifying the ensuing discussion, the following example limits the processing to the intermediate code of innermost loops. Of course, it is easy to process other intermediate code through my inventive method. Nevertheless, since most of the execution time for an object program is consumed in executing the innermost loops in that program, then, in almost all cases, processing only these innermost loops through my inventive method is sufficient. Hence, I will explain the processing undertaken by the three steps of cache conflict reduction process 20 in the context of the innermost loops.

In particular, within process 20, conflict analysis process 30 first detects whether intermediate code 5 would generate cache conflicts and then analyzes the conditions which would generate each conflict. The detected instances of cache conflicts, i.e., "cache conflict generation detection results", provided by process 30 are used in the following step, i.e., decision step 40. Cache conflict generation conditions are provided, by process 30, as cache conflict information 9 for subsequent use by memory reference reordering process 50.

Based on the results of cache conflict analysis 30, i.e., whether a cache conflict would be generated by intermediate code 5, decision step 40 determines whether memory reference reordering process 50 will then be executed. In particular, when such a cache conflict would be generated, as detected by cache conflict analysis process **30**, a cache conflict is said to exist. Hence, decision block **40** routes execution to process **50** which, in turn, reorders the memory references to reduce, if not eliminate, these conflicts. Alternatively, if analysis process **30** finds that no cache conflict would be generated within intermediate code **5**, decision block **40** routes execution around memory reference reordering process **50**, hence skipping any such reordering, thus effectively terminating further processing of intermediate code **5**.

Through memory reference reordering process **50**, the order through which memory references would be carried out is changed based on cache conflict information **9** in order to reduce cache conflict. In essence, when the memory reference order is changed, the resulting order is such that all references to a memory block previously transferred to a cache block are completed before other conflicting memory references occur to the memory block; thus subsequently preventing the memory block from being rejected from cache.

With the above in mind, I will next describe the structure of data input and output for each step within cache conflict reduction process **20**. Inasmuch as the specific structures of source program **3** and object program **7** are not related to my inventive process **20**, these structures are omitted. However, the structure of intermediate code **5** and cache conflict information **9** will be explained.

Specifically, intermediate code **5** is an internal compiled representation of a source program, the representation being similar to that shown in FIG. **2**. In this preferred embodiment, a structure known as a three address statement is used for intermediate code **5**. Three address statements are described in A. V. Aho and J. D. Ullman, *Compilers* (©1985, Addison Wesley Publishing Co.), pages 464–473. Two columns of data are required for use with cache conflict reduction process **20**. In that regard, a statement number (**305** shown in FIG. **16**) column and reference group number (**330**) column are used along with the three address statements.

FIG. **16** shows an example of intermediate code **5**. As shown, intermediate code **5** corresponds to source program **3**, the latter shown in FIG. **2**. As shown in FIG. **16**, the statements in intermediate code **5** are lined up in execution order. Each of these statements is composed of: statement number **305**, which starts at 1 and increases incrementally; label **310** which specifies a branch target; three address statement **320** and, as shown, its contents—all of such steps implementing the innermost loop in source program **3** (see FIG. **2**); and reference group number **330** shown in FIG. **16** which is used as a work area by cache conflict reduction process **20**. Label **310** is not used other than as a branch target. Reference group number **330** is added only to memory references; nothing is present in this column at the input to cache conflict reduction process **20**.

By using a three address statement as the format of each statement in intermediate code **5**, each memory reference is separated into individual statements. As such, it is easy to add such statements or change the order of memory reference statements. Though my inventive method can be implemented using other formats, than a three address statement, for intermediate code **5**, accompanying processing does become increasingly complex.

It is sufficient to convert the source program to appropriate three address statements immediately prior to performing cache conflict reduction process **20**. Hence, when performing optimization processing between syntax analysis process **10** (as shown in FIG. **1**) and cache conflict reduction process **20**, another format can be used for optimization processing. In that regard, a three address statement can be represented by, e.g., a syntax tree generally used for optimization processing; hence, the format of intermediate code **5** can be one consistent with a syntax tree.

Cache conflict information **9**, as shown in FIG. **1**, specifies the cache conflict generation conditions that result from memory references of those statements in intermediate code **5**. Using FIG. **17**, I will now explain the structure of cache conflict information **9**.

As shown in FIG. **17**, cache conflict information **9** is constructed from cache reference group information **9900** and cache conflict graph **9500**. Cache reference group information **9900** is a summary of memory references to common memory blocks and consists of cache reference groups, only one of which, i.e., group **9010**, is generally shown. These groups, in turn, are then classified as described below. Cache conflict graph **9500** shows the cache conflict generation conditions between cache reference groups. In other words, cache reference group information **9900** is a collection of cache reference groups, and each cache reference group, e.g., group **9010**, is a collection of statements which perform memory references to the same memory block in intermediate code **5**.

Specifically, each cache reference group, such as group **9010**, is composed of a cache reference group number, e.g., number **9020**, and for each memory reference statement in that group: a statement number (**9040** denoting all such statement numbers in this group), a referenced array element (**9060** denoting all such array elements in this group), and an address calculation expression (**9090** denoting all such expressions in this group) for corresponding referenced array element **9060**.

With the group number of cache reference group **9010** as a node, cache conflict graph **9500** has edges (lines) between each pair of separate cache reference groups which generate each cache conflict. Having a cache conflict generated between two cache reference groups means that the memory blocks which reference the memory reference statements in each cache conflict group, e.g., group **9010**, are mapped into the same cache block. As specifically shown in FIG. **17**, nodes **9510** and **9520** of cache conflict graph **9500** are connected by edge **9590**; hence, this figure shows that a cache conflict is generated between two cache reference groups corresponding to these two nodes.

I have now completed my summary explanation of the inventive compiler shown in FIG. **1**. Accordingly, I will proceed to explain, in detail and in conjunction with a specific example, the processing undertaken by cache conflict analysis process **30**, and later in the context of this example, the ensuing results, and thereafter memory reference reordering process **50**—both of these steps being contained within cache conflict reduction process **20** shown in FIG. **1**.

I will begin by explaining, in detail, cache conflict analysis process **30**. Through process **30**, memory references in intermediate code **5** such as those shown in FIG. **16** are analyzed, and resulting cache conflict information **9**, as shown in FIG. **17**, is generated.

FIG. **8** shows a high-level flowchart of the processing procedure of cache conflict analysis process **30** shown in FIG. **1**.

As shown in FIG. **8**, upon entry into process **30**, execution proceeds to cache reference group analysis step **32**. This latter step, through analyzing the memory reference statements in intermediate code **5** (shown in FIG. **16**), summarizes references to memory blocks as cache reference groups (such as group **9010** shown in FIG. **17**), and generates cache reference group information **9900**. Next, cache conflict graph analysis step **34** is executed which, based on cache reference group information **9900**, analyzes cache conflict conditions between different cache reference groups, and generates cache conflict graph **9500** in, e.g., graphical form, which represents the results of this analysis.

Thereafter, decision step **36**, when executed, determines whether or not an edge exists in cache conflict graph **9500**, i.e., whether this graph shows a conflict between two cache reference groups. If such an edge exists, decision step **38** routes execution, via its true (T) path, onward to step **38**, at which point processing of step **30** ends. Alternatively, if no such edge exists, decision step **38** routes execution, via its false (F) path, onward to step **39** at which point no cache conflict is said to exist and processing ends. The result, if true, from process **30**, specifically the fact that an edge exists in graph **9500**, is passed onward to decision step **40** shown in FIG. **1**.

Next, using FIGS. 9A–12, I will explain, in detail, the processing undertaken by cache reference group analysis step **32** and cache conflict graph analysis step **34** shown in FIG. **8**.

FIGS. **9A-9B** collectively show a detailed flowchart of cache reference group analysis step **32**; the correct orientation of the drawing sheets for these figures is shown in FIG. **9**. As explained above, through execution of cache reference group analysis step **32**, cache reference group information **9900** is generated from intermediate code **5**.

Upon entry into step **32**, as shown in FIGS. 9A-9B, through execution of step **3202** cache reference group number n is initialized to zero. Steps **3204** and **3234** represent a "repeat" ("looping") operation. Processing is done, within step **32**, in order from the first statement to the last statement of intermediate code **5**, with the current statement being processed in code **5** being denoted by variable Si.

Thereafter, decision step **3206**, when executed, determines whether statement Si references memory, i.e., whether that statement is, among other things, a memory reference statement. If this statement references memory, execution proceeds, via the true (T) path from this decision step to decision step **3208**. Alternatively, if statement Si does not reference memory, then this statement is not processed further by cache reference group analysis step **32** and execution proceeds, via the false (F) path emanating from decision step **3206**, onward to step **3234**. Decision step **3208**, when executed, determines whether statement Si has been registered to (associated with) any cache reference group, e.g., group **9010**. If no such registration has occurred, decision step **3208** routes execution, via its false path, forward to step **3210**. Alternatively, if statement Si has already been registered to any cache reference group, then no further processing occurs within step **32** for this statement and execution proceeds, via the true path emanating from decision step **3208**, onward to step **3234** to process the next statement.

Step **3210**, when executed, increments number n by 1 and then sets variable m to the current n value. Variable m is the initial value of the cache reference group number for the corresponding cache reference group that will be processed in the following steps. Cases exist where the value of variable m changes while processing of a cache reference group is in progress.

After step **3210** has been executed, step **3212** is executed to establish cache reference group CAG(m), for which the cache reference group number is m, and to register statement Si to this cache reference group, i.e., CAG(m). As shown in FIG. 17, when statement Si is registered, a statement number (**9040**), a referenced array element (**9060**), and an address calculation expression are written for this statement to cache reference group information **9900**, and specifically to an entry for cache reference group CAG(m). Because the method for obtaining the address calculation expression (**9090**) from the referenced array element (**9060**) is widely known, it will be omitted here.

Next, as shown in FIGS. 9A and 9B, steps **3214** to **3232** are repetitively performed to process the statement that immediately follows statement Si, i.e., statement S(i+1), until the last statement. For ease of reference, the particular statement currently being processed by these steps is denoted as statement Sj.

Decision step **3216**, when executed, determines whether statement Sj references memory, i.e., whether this statement is, among other things, a memory reference statement. If statement Sj references memory, execution is routed by this decision block, via its true path, onward to step **3218**. Alternatively, if statement Sj does not reference memory, then this statement is not processed further with execution routed, via a false path emanating from decision block **3216**, onward to step **3232**.

When executed, step **3218** calculates a distance, d, between memory reference statement Sj and cache reference group CAG(m). The distance between memory reference x and cache reference group y is a minimum value of distance in main memory between memory reference x and all the memory references registered to cache reference group y. The calculation method for this distance is explained in detail later through FIG. **10**.

Once step **3218**, shown in FIGS. 9A and 9B, is completed, execution proceeds to decision step **3220**, which determines whether distance d is less than or equal to a designated value Dmax. If distance d is less than or equal to Dmax, decision step **3220** routes execution, via its true path, onward to decision step **3222**. Alternatively, if distance d exceeds Dmax, then statement Sj is not processed further with decision block **3220** routing execution, via its false path, to step **3232**.

Designated value Dmax is a "threshold value" for determining whether or not memory references refer to the same memory block and is based on the distance between memory references in main memory. This threshold value should be set equal to ("the memory block length")/2. If distance d is less than or equal to the threshold value Dmax, then both memory references basically refer to the same memory block. As described below and with reference to this preferred embodiment, it has been determined that Dmax is preferably equal to 4. However, Dmax is not so limited as long as the value chosen for Dmax is one which can be used to determine whether or not two memory references refer to the same memory block.

Consequently, in the preferred embodiment, decision block **3220** determines whether or not memory reference statement Sj and the memory reference in cache reference group CAG(m) refer to the same memory block.

An alternate method for determining whether or not memory references refer to the same memory block relies on directly obtaining the referenced memory block from the position in main memory of the referenced array element. However, here, since the array element address changes depending on the loop iteration, one must necessarily and sufficiently consider that the referenced memory block also changes; hence, processing becomes somewhat complex. Nevertheless, this alternate method, if sufficiently expanded in this way, could be used with the preferred embodiment.

Decision step **3222**, when executed, determines whether statement Sj is registered to any of the existing cache reference groups, e.g., group **9010**. If statement Sj is not so registered, then this decision block routes execution, via its false path, to step **3224**. Through execution of step **3224**, processing statement Sj is registered to cache reference group CAG(m), from which step execution proceeds forward to step **3232**. Alternatively, if decision block **3222** determines that statement Sj is registered in an existent cache reference group, then this decision block routes execution, via its true path, to block **3226**.

Through execution of steps **3226** to **3230**, the memory references in cache reference group CAG(m) are merged with existing cache reference groups. First, step **3226**, when executed, re-registers all memory references currently registered in cache reference group CAG(m), into the one cache reference group, of groups **9010**, to which statement Sj belongs. Next, through execution of step **3228**, value n is decremented by one, and cache reference group CAG(m) is eliminated. Thereafter, through execution of step **3230**, the cache reference group number of Sj is set to m.

Steps **3232** and **3234** represent ends of repeat (loop) processing; step **3236** represents the end of execution of cache reference group analysis step **32**.

Next, I will describe the processing inherent in step **3218** mentioned above, specifically, the calculation of the distance d between the memory references of processing statement Sj and cache reference group CAG(m).

FIG. **10** shows a detailed flowchart of distance calculation step **3218**. Upon entry into step **3218**, execution proceeds to step **3252**, which obtains the address calculation expression AEj for the memory referenced by statement Sj. The address calculation expression AEj can be easily obtained from array elements specified by memory references. Since this is a well known method, I will omit all detailed explanation thereof.

Thereafter, execution proceeds to step **3254** which sets the initial value of d equal to Dmax+1. This is followed by repetitive execution of steps **3256** to **3268**. Steps **3256** to **3268** are executed sequentially for all memory references Ai registered in cache reference group CAG(m) which is one of the cache reference groups **9010**.

Through execution of step **3258**, the address calculation expression, AEi, **9090** of memory reference Ai is extracted from cache reference group CAG(m). Thereafter, step **3260** is executed to calculate a difference, diff, between address calculation expressions AEi and AEj. Inasmuch as step **3260** is implemented with conventional, and simple symbolic expression processing, I will omit all details here. Consequently, through the above described processing, the address difference for the memory references is obtained.

Next, decision step **3262** is executed to determine whether difference diff is a constant. If this difference is a constant, execution proceeds, via the true path emanating from this decision step, to decision step **3264**. Alternatively, if difference diff is not a constant, then this difference is not processed further by steps **3264** to **3268** and execution proceeds, via the false path emanating from decision step **3262**, to step **3272**. Execution of step **3272** sets distance d equal to the current difference value diff, with execution thereafter proceeding to final step **3270** signifying completion of distance calculation step **3218**.

Decision step **3264**, when executed, determines whether an absolute value, abs, of the difference diff is smaller than distance d. If the absolute value is smaller in that regard, then decision step **3264** routes execution, via its true path, to step **3266**, wherein, when executed, the absolute value of difference diff is set as the new value for distance d. Alternatively, if decision step **3264** determines that the absolute value exceeds or equals distance d, then this decision step routes execution, via its false path, directly to step **3268**, and hence, back to step **3256**, thereby skipping step **3266**.

Based on the above processing, difference d is set to be a minimum value of the distance in memory between the memory references of statement Sj and all memory references in cache reference group CAG(m). However, when this distance is not a constant, distance d remains set, through step **3272**, as the expression which shows the distance between two memory references in this group.

When the distance d is an expression, the size relationship between d and Dmax is unclear. Hence, by using the value of the distance d set in step **3218**, the result of the comparison ($d \leqq Dmax$?) undertaken by decision step **3220** in FIGS. 9A and 9B becomes false. As such, the memory references of statement Sj are not mapped into the same cache block as are the memory references in cache reference group CAG (m).

This concludes my explanation of cache reference group analysis step **32**. With the above described processing, cache reference group information **9900**, encompassing all the cache reference groups, is generated.

Next, I will explain, in detail, the processing inherent in cache conflict graph analysis step **34** shown in FIG. **8**.

FIG. **11** shows a detailed flowchart of cache conflict graph analysis step **34**. This step generates cache conflict graph **9500** based on cache reference group information **9900** (both of which are shown in FIG. **17**).

As shown in FIG. **11**, upon entry into step **34**, step **3402** is executed to generate the nodes of cache conflict graph **9500**. A node is made for each cache reference group; the cache reference group number, such as number **9020**, of each cache reference group, such as group **9010**, is also added by this step.

Thereafter, through repetitive execution of steps **3404** to **3418**, the edges (lines) between these newly created nodes are established. These edges are set between the cache reference groups, e.g. group **9010**, that have been determined, in the manner described above, as generating cache conflict. In particular, steps **3404** to **3418** are repeated for each value of i, incrementing by one from 1 to "the number of cache reference groups". Steps **3406** to **3416** are also repeated, but for each value of i, incrementing by one from i+1 to "the number of cache reference groups".

Specifically, through execution of step **3408**, distance dc within cache **110** (see FIG. 29) and between cache reference groups CAG(i) and CAG(j) is determined. The method for obtaining distance dc will be explained, in detail, below with reference to FIG. 12.

Thereafter, decision step **3410** shown in FIG. **11**, when executed, determines whether distance dc is less than or equal to designated value DCmax. If this inequality is satisfied, decision step **3410** routes execution, via its true path, onward to decision step **3412**. Alternatively, if distance dc exceeds value DCmax, then steps **3412** and **3414** are skipped, with decision block **3410** routing execution, via its false path, to step **3416**.

Designated value DCmax is a threshold value for determining whether or not cache conflict is generated based on the distance between memory references in cache. Suitable values for DCmax are from 0 to "cache block length". Hereafter, I assume DCmax to have the value 4. However, the threshold DCmax value is not limited to those given above and can be any value which determines whether or not cache conflict is generated. Hence, in this preferred embodiment, as described above, a determination is made whether or not memory blocks of memory references in cache reference group CAG(i) and memory references in cache reference group CAG(j) are mapped to the same cache block, in other words, whether or not these memory references cause cache conflict with each other.

Another method that can be alternatively used to determine whether or not memory blocks are mapped to the same cache block involves directly obtaining cache blocks mapped from the position of the reference array elements in main memory. However, in this case, the array element addresses change with loop iterations. Therefore, it is important to take into consideration that mapped cache blocks also change, which, in turn, complicates attendant processing. Nevertheless, this alternate method can be used in the preferred embodiment of my invention.

Now, continuing with the discussion of step **34** shown in FIG. **11**, decision step **3412**, when executed, determines whether an edge has been established, i.e., already exists, between nodes i and j in cache conflict graph **9500**. If such an edge does not yet exist in the graph, then decision step **3412** routes execution, via its false path, to step **3414**, which, in turn, sets (places) an edge in the graph between nodes i and j. Execution then loops back, if appropriate, as noted above, via steps **3416** and **3406**, to step **3408**, and so on. Alternatively, if this particular edge already exists in the graph, then decision step **3412** routes execution, via its true path, directly to step **3416** from which execution loops back, if appropriate and as noted above, to step **3408**, and so on; else execution of step **34** terminates at step **3420**.

Next, I will explain the processing inherent step **3408** which as described above, determines distance dc on cache **110** between cache reference groups CAG(i) and CAG(j).

FIG. **12** shows a detailed flowchart for step **3408**. Upon entry into step **3408**, execution proceeds to step **3452** which, when executed, sets an initial value for distance dc as equaling DCmax+1. Steps **3454** to **3466** are then repetitively executed for each and every memory reference Ai registered to cache reference group CAG(i).

Through execution of step **3456**, distance diffc between memory reference Ai and cache reference group CAG(j) is obtained. Since virtually the same processing is used in this step as is used to implement step **3218**, explained above in conjunction with FIG. **10**, for the sake of brevity, I will omit any further details of this processing. After step **3456** has fully executed to determine distance diffc, step **3458** is executed to determine whether distance diffc is a constant. If this distance is a constant, then decision step **3458** routes execution, via its true path, onward to step **3460**. Alternatively, if distance diffc is not a constant, then decision step **3458** routes execution, via its false path, to step **3470**.

Through execution of step **3460**, a modulus, i.e., remainder of a quotient, of distance diffc divided by the cache size is calculated, with the distance diffc then being set to a value of the remainder. This calculation provides the distance on cache **110**. Next, decision step **3462** is executed to determine whether an absolute value of distance diffc is less than distance dc. If the absolute value is less than the current value of distance dc, then decision step **3462** routes execution, via its true path to step **3464**, which sets distance dc equal to this absolute value. Alternatively, if decision step **3462** determines that the absolute value of distance diffc exceeds or equals distance dc, then step **3464** is skipped with execution proceeding, via the false path emanating from this decision block, to step **3466**. Step **3470** when executed, sets distance dc equal to the expression for distance diffc. Thereafter, execution loops back, if appropriate and as noted above, via steps **3466** and **3454**, to step **3456**, and so on; else, execution of step **3408** terminates at step **3468**.

Once step **3468** is reached, execution of cache conflict graph analysis step **34** ends, with cache conflict graph **9500** being generated. Hence, as explained above in conjunction with FIG. **8**, at this point in the processing, decision step **36** is executed to determine whether an edge, hence indicative of a cache conflict, exists in cache conflict graph **9500**.

FIG. **18** shows the contents of cache conflict information **9** after cache conflict analysis process **30** (as shown in FIGS. **8** to **12** and as described above) is performed on intermediate code **5** (discussed above and shown in FIG. **16**) of source program **3** shown in FIG. **2**. As shown in FIG. **18**, cache conflict graph **9500** contains a graph consisting of nodes **951**, **952** and **955**, and a graph consisting of nodes **953** and **954**. Cache reference group information **9900** includes multiple cache reference groups **9010**, **9110**, **9210**, **9310**, and **9410**, with corresponding cache reference group numbers **9020** to **9024** having been added to the respective cache reference groups.

From cache conflict graph **9500** shown in FIG. **18**, cache conflict occurs between cache reference groups **9010**, **9110**, and **9410** with the cache reference group numbers **1**, **2** and **5**, respectively. Also, this graph indicates that cache conflict occurs between cache reference groups **9210** and **9310** with the cache reference group numbers **3** and **4**, respectively.

Cache reference group **9010** is made from two memory references A(I,J) and A(I+1,J), each of the other reference groups **9110**, **9210**, **9310**, **9410** consists of only one memory reference. The reason that A(I,J) and A(I+1,J) are both in the same cache reference group **9010** is that, for cache reference group analysis step **32** (shown in FIG. **8**), the difference between the address calculation expressions for these memory references as given in FIG. **18**; namely, "(the starting address of A)+4*(I−1+256*(J−1))" and "(the starting address of A)+4*(I+1−1+256*(J−1))" is −4. Here, the absolute value is less than or equal to Dmax (in this case, as noted earlier, Dmax is taken to be 4).

The reason that an edge exists between nodes **952** and **951** in cache conflict graph **9500** is as follows. In cache conflict graph analysis step **34** (shown in FIG. **8**), the difference between respective address calculation expressions, given in FIG. **18**, "(the starting address of B)+4*(I−1+256*(J−1)+256*128*(2−1))" and "(the starting address of A)+4*(I−1+256*(J−1))" of memory references B(I,J,2) and A(I,J), respectively, in cache reference groups **9210** and **9010** corresponding to the above node numbers is "4*256*128+(the starting address of B—the starting address of A)". In comparison, as shown in FIG. **3**, for contiguous storage, the starting address for array B is situated 128Kbytes after the starting address of array A, so the above mentioned difference becomes 256Kbytes, the remainder after division by cache size (256Kbytes) is 0, and the absolute value (0 in this case) is clearly less than Dmax (DCmax=4 in this case). Inasmuch as the remaining edges in FIG. **18** are derived in the same similar manner as that for the edge between nodes **951** and **952**, for brevity I will omit any such discussion for all the remaining edges in cache conflict graph **9500**. Hence, I will not explain other portions of FIG. **18**.

As described above, after the cache conflict information is generated, through execution of cache conflict analysis process **30**, and cache conflicts have been verified as existing through execution of decision step **40** shown in FIG. **1**, memory reference reordering process **50** is then performed. Having fully explained process **30** and decision step **40**, I will now explain, in detail, reference reordering process **50**. Through execution of process **50**, cache conflicts are reduced by changing the order of memory references so that, after re-ordering, all references to a memory block, previously transferred to a cache block, are completed before other memory references to that memory block occur; thus preventing the memory block from being rejected from cache **110**.

FIG. **13** is a high-level flowchart which provides an overview of the processing inherent in memory reference reordering process **50**. Upon entry into process **50**, execution proceeds to step **51**, which, when executed, writes cache reference group numbers, e.g., number **9020**, for intermediate code **5** into reference group numbers **330** (see FIG. **19**) of memory reference statements but only for those cache reference groups, e.g., group **9010**, which cause cache conflicts according to cache conflict information **9**. For example, based on the cache conflict information **9** shown in FIG. **18**, once intermediate code **5**, shown in FIG. **16**, has been processed to include the corresponding cache reference group numbers as reference group numbers **330**, the resulting intermediate code is shown in FIG. **19**.

Returning to FIG. **13**, once step **51** has fully executed to fully write all the corresponding cache reference group numbers, step **52** is executed. Step **52**, based on cache conflict information **9**, determines a particular loop unrolling method which will be performed in a next step, i.e., step **53**. Loop unrolling step **53** is performed as part of memory reference reordering process **50** because I anticipate that memory references to the same memory block will occur over multiple loop iterations. Through use of loop unrolling step **53**, multiple changes in the order of memory references can be undertaken over multiple loop iterations, so that overall cache reduction process **20**, of which reordering process **50** is a part, maximally reduces cache conflicts. Hence, for optimal cache conflict reduction, step **52** is used to determine the optimal loop unrolling method. I will later explain, in detail, how step **52** determines the proper loop unrolling method.

For example, based on cache conflict information **9** shown in FIG. **18**, and for intermediate code **5** of FIG. **19**, optimal loop unrolling involves unrolling the innermost loop of I four times. In that regard, step **52** determines the parameters of unrolling, i.e., which loop to unroll and the number of times the loop should be unrolled.

Once the optimal loop unrolling method has been determined, execution proceeds to step **53**, which, when executed actually unrolls the loop according to these loop parameters. Once the pertinent parameters, such as those set forth in the example immediately above, are determined by step **52**, the actual process itself of unrolling a loop is well known. Hence, I will not explain loop unrolling in any further detail.

After step **53** fully is executed, step **54** is executed to process intermediate code **5** which was generated by the loop unrolling step **53**. Specifically, through execution of step **54**, multiple assignment statements having the same target variable(s) in this code are detected, and if it is possible to rename the variable(s), the variable(s) is renamed. Through step **54**, other than when necessary, the

target variables of assignment statements are changed so as to be different from each other. Doing so increases the freedom to change the order of the intermediate code, and allows for more optimal reordering to be subsequently undertaken. Conversely, if an assignment statement for the same variable is unchanged, i.e., left as is, these assignment statements can not be reordered with respect to each other, thereby increasing the difficulty through which the intermediate code can be reordered. Since the variable renaming process performed by step **54** is also well known, I will not explain it in any further detail here.

For example, if the loop unrolling step **53** and variable renaming step **54** are executed on intermediate code **5** shown in FIG. **19**, resulting intermediate code **5** as shown in FIGS. **20A** and **20B** is obtained; for which the correct orientation of the drawing sheets for these figures is shown in FIG. **20**. As explained above, given the determination of the loop unrolling method, i.e., the pertinent parameters therefor, determined in step **52** of illustratively, unrolling the innermost loop of I four times, then, as shown in FIGS. **20A** and **20B**, the innermost loop has been unrolled by 4 times for I. Where possible, variables, existing in code **5** shown in FIG. **19**, have been renamed in the code shown in FIGS. **20A** and **20B**.

After variable renaming is completed, execution proceeds to step **55** as shown in FIG. **13**, which, when executed, eliminates references to the same array elements in intermediate code **5** as this code then exists. As a result, redundant memory references are eliminated. Since the elimination process undertaken by step **55** is also widely known, for brevity I will omit any further explanation of it.

For example, when the elimination of common array elements is performed through execution step **55** on intermediate code **5** shown in FIGS. **20A** and **20B**, intermediate code **5** such as that shown in FIGS. **21A** and **21B** is obtained; the correct orientation of the drawing sheets for FIGS. **21A** and **21B** is shown in FIG. **21**. For the illustrative intermediate code shown in FIGS. **20A** and **20B**, statements **11**, **21** and **31** as identified by statement number **305**, have all been eliminated thereby yielding intermediate code **5** shown in FIGS. **21A** and **21B**. Whenever loop unrolling is performed, several redundant array references are generated as shown in FIGS. **20A** and **20B**, thereby necessitating the elimination of these references through step **55**.

Finally, as shown in FIG. **13**, step **57** is executed. Based on reference group numbers **330** written to cache conflict information **9** and as well to intermediate code **5**, the intermediate code, that resulted from execution of step **55**, such as that shown in FIGS. **21A** and **21B**, is reordered through step **57** to reduce cache conflicts. Specifically, the order of memory references is changed so that, after re-ordering, all references to a memory block, previously transferred to a cache block, are completed before other memory references occur to that memory block; thus preventing the memory block from being rejected from cache **110**.

I have now explained, in an overview summary fashion, the processing inherent in memory reference reordering process **50**.

Though I have shown and described the best implementation for the preferred embodiment of my invention, I realize that memory reference reordering process **50** includes many processing steps. Nevertheless, some degree of cache conflict reduction could be achieved, if in lieu of using my entire implementation of process **50**, i.e., using loop unrolling method determination step **52** and loop unrolling step **53**, variable renaming step **54** and common memory array element elimination step **55**, only one or a sub-set of these steps were to be used instead.

For example, if source program **3** were to be provided with its innermost loop having been sufficiently unrolled, then acceptable performance would result even if either loop unrolling method determination step **52** or loop unrolling step **53** was not performed. Conversely, for source programs which are provided with absolutely no loop unrolling, then these steps are essential and must be performed. Hence, as will be plainly evident to one skilled in the art, many such partial embodiments of my inventive method can be devised and implemented, but for brevity I will not describe them in any further detail here.

Inasmuch as intermediate code **5** experiences a variety of successive changes as the code progresses from loop unrolling step **52** through common array element elimination step **55** as shown, e.g., in FIG. **13**, then, prior to realigning the intermediate code in step **57**, one can alternatively re-execute cache conflict analysis process **30** (see FIG. **1**), including step **51** (see FIG. **13**) in which cache reference group numbers, e.g., number **9010**, are written to intermediate code **5**, just prior to executing step **57**. Since the effectiveness of cache conflict reduction process **20** does not markedly change, i.e., increase by much, I see no need to explain this alternate technique in any further detail.

Now, returning to memory reference reordering process **50** shown in FIG. **13**, I will now describe, in detail, both loop unrolling method determination step **52** and intermediate code reordering step **57**. First, I will explain the loop unrolling method determination step **52**.

FIG. **14** is a detailed flowchart showing the loop unrolling method determination step **52** shown in FIG. **13**.

First, upon entry into step **52** as shown in FIG. **14**, step **521** is executed which, using cache conflict information **9**, generates loop unrolling candidate table **8**. This table, when fabricated, contains all loop unrolling candidates from which the loop(s) that is ultimately unrolled is selected.

FIG. **22A** shows the structure of loop unrolling candidate table **8**. In this table, for each separate cache reference group, such as group **9010** shown in FIG. **18**, that causes a cache conflict, the table maintains a separate entry containing a cache reference group number, **82** shown in FIG. **22A**, a suitable unrolling loop candidate **84**, and a loop unrolling count candidate **86**. After loop unrolling candidate table **8** is generated through execution of step **521**, thereafter step **536**, as shown in FIG. **14**, is executed to extract the entry for the candidate loop to be unrolled which appears most often in this table. The loop specified in this entry becomes the loop that is selected to be unrolled. Thereafter, step **538** is executed to extract the largest value of the unrolling count candidate **86** entries stored for the selected loop; this largest value becomes the loop unrolling count, i.e., the number of times this selected loop is to be unrolled.

From the above, the optimal loop unrolling method, i.e., the loop unrolling parameters, is determined for cache conflict reduction process **20**.

With the above in mind, I will now proceed to explain, in detail, loop unrolling candidate table generation step **521**. During execution of loop unrolling candidate table generation step **521**, steps **522** to **534** are separately repeated for each different cache reference group CAG(i).

Specifically, upon entry into step **521**, execution proceeds, via repeat step **522**, to decision step **524**. Using cache conflict information **9**, decision step **524** determines whether cache conflict exists between cache reference group CAG(i) and any other cache reference group, e.g., group **9010**. If such a cache conflict exists, then decision step **524** routes execution, via its true path, to step **526**. Alternatively, if no such cache conflict exists, then steps **526** through **532** are skipped, with decision step **524** routing execution, via its false path, to repeat step **534**, to assess cache conflict for the next cache reference group CAG(i+1), or if all such groups have been processed, proceed to step **536**.

Through execution of step **526**, one memory reference in cache reference group CAG(i) is selected, with the address calculation expression **9090** (see FIG. **17**) therefor being assigned to variable AE. Next, step **528** is executed to determine the unrolling loop candidate entry **84**. Specifically, for the possible loops to be unrolled, the stride for each loop is determined. For any loop, the stride represents a distance, in memory addresses and in absolute value terms, between a memory reference in one iteration of the loop, i.e., at address AE(ii), and the same reference in the next iteration of that loop, i.e., at address AE(ii+inc), with variable inc defining an increment of the loop from loop iteration ii. The loop having the lowest stride for the one memory reference in CAG(i) is selected as a candidate loop to be unrolled for cache reference group CAG(i).

Next, step **530**, which when executed, selects the candidate loop unrolling count, i.e., the number of times a loop is to be unrolled, for the candidate loop selected through step **528** for cache reference group CAG(i). Specifically, the candidate unrolling count is selected, for the candidate loop, as a maximum value of all unrolling counts for that loop such that, after this loop is unrolled, all the memory references in CAG(i) are included within one memory block. Thereafter, step **532** is executed to write an entry for cache reference group CAG(i) into table **8** (see FIG. **22B**) containing a cache reference group number **82**, and, for this group, the loop to be unrolled for this group, i.e., (optimal) unrolling loop candidate **84**, and loop unrolling count **86** therefor.

FIG. **22B** shows the loop unrolling candidate table **8** generated through the above-described procedure for cache conflict information **9** shown in FIG. **18**. The particular loop "DO **20** I=1, 255" appears most frequently as unrolling loop candidate **84**, with the largest loop unrolling count candidate **86** for this loop being "4". Once the loop processing for all the cache reference groups is completed and table **8** is fabricated as shown in FIG. **22B**, then, given the illustrative contents of table **8**, through execution of steps **536** and **538** shown in FIG. **14**, this particular loop is selected as the loop to be unrolled, with its corresponding loop unrolling count stored in table **8** specifying the number of times, i.e., the unrolling count, this loop is to be unrolled.

Next, I will describe, in detail, as part of cache conflict reduction process **50** (see FIG. **1**), reordering of intermediate code step **57** (hereinafter, for simplicity, referred to as "intermediate code reordering" step **57**) shown in FIG. **13**.

FIG. **15** is a detailed flowchart of intermediate code reordering step **57**. The operations undertaken by step **57** are based on "list rescheduling", which is a conventional process of generally reordering instructions to increase overall program execution speed. List rescheduling is discussed in references, such as *ACM SIGPLAN Notices,* Vol. 25, No. 7, July 1990, pages 97–106.

Clearly, other methods of reordering instructions other than that specifically used in the preferred embodiment can be employed. However, to reduce cache conflict for a given memory block, the order of memory references must be adjusted before all referencing, to memory blocks previously transferred, to a cache block would have been complete, so that the given memory block will not be rejected from cache. To do this, in this preferred embodiment, selection processing has been added midway through the processing of step **57** in form of scheduling instructions—which, as described below, are provided by step **584**. Furthermore, steps **582**, **586** and **590** are used, as described below, to ensure proper operation in those instances where memory reference reordering can not be accomplished.

In particular, intermediate code reordering step **57** is formed of two major steps.

First, through execution of step **570** shown in FIG. **15**, each statement of intermediate code **5** is allocated to schedule table **26**. Schedule table **26** shows execution statements for each time slot of program execution; this table has the format shown in FIGS. **24A** and **24B**; the correct orientation of the drawing sheets for these figures being shown in FIG. **24**. Each line in this table is composed of time slot value **303**, statement number **305**, three address statement **320** and reference group number **330**. When multiple statements are executed in one time slot, schedule **26** is constructed of columns **305**, **320** and **330** repeated the appropriate number of times equal to the number of statements so executed in that slot.

Second, after step **570** has allocated each statement of intermediate code **5** to schedule table **26**, step **598** is executed to reorder intermediate code **5** according to the order within schedule table **26**.

Given that overview of step **57**, I will explain, in detail, the processing inherent in step **570** to allocate each statement of intermediate code **5** to the schedule table.

In particular, upon entry into step **570**, execution proceeds to step **572** which, when executed, generates data dependency graph **24** for intermediate code **5** as it then exists. FIG. **23** shows illustrative data dependence graph **24** for intermediate code **5** shown in FIGS. **21A** and **21B**. This particular graph depicts the execution order inter-relationships for the statements in intermediate code **5**. Once this graph is constructed through step **572**, step **574**, shown in FIG. **15**, is executed to prioritize, through a critical path method, the nodes in data graph **24**.

Graph **24** shows the execution order inter-relationships between each statement of intermediate code **5** represented as a node, and an arc(s) connecting each such node to those which need to precede or follow that node, i.e., the corresponding statements, in the execution order. Additional data has been added to each node and arc. For example, for any statement represented by a node, a number on the left side in a field of a node, e.g., node **242**, is statement number **305** for the statement corresponding to that node, and the statement on the right side is the corresponding three address statement **320**. The numbers added to the right of each node are the priority and reference group number **330**, when the statement is executed. However, reference group number **330** is only added to statements that include memory references that cause cache conflict.

As an example, arc **244** represents that the statement of node **242** must be executed before the statement of node **246** is executed. The numbers added to the arc show the duration of execution of the preceding statement, such as "2" in the case of the arc connecting these two nodes. Priority through the critical path method is determined by a duration of the longest path from each node to a final node.

After step **574**, shown in FIG. **15**, has fully executed to determine priority for all the arcs in the graph, step **576** is executed to initialize variable t, which specifies the current time slot, to 1. Thereafter, step **578** is executed, which, for statements that have not yet been allocated to the schedule table, detects a set of memory reference statements (SRs) which can be executed in time slot t. However, a statement that is able to execute at time slot t means that all immediately preceding statements, having been allocated to the schedule table, must have completed their execution before time slot t.

Next, execution proceeds to decision step **580**. This step determines whether or not the set of SRs produced by step **578** is empty. If this set of SRs is empty, i.e., no statements are seen as being executable in time slot t, then decision step **580** routes execution, via its true path, to decision step **592**. Alternatively, if this set of SRs is not empty, decision step **580** routes execution, via its false path, to step **582** which, in turn, saves the current contents of this set to set SRorig.

Once step **582** has executed, execution proceeds to step **584**. The latter step removes from the set of SRs the intermediate code for memory reference statements that belong to cache reference groups which conflict with other cache reference groups and for which only a portion of those statements is allocated to the schedule table. Through step **584**, the order of memory references is controlled, i.e., specifically re-ordered, to reduce cache conflict before all referencing to a memory block previously transferred to a cache block would have been completed, so that the memory block is not rejected from cache.

Specifically, through step **584**, allocation of the memory reference statements, that have been removed from the set of SRs, to the schedule table has not been stopped, but rather, just delayed. In particular, each memory reference statement that has been removed from the set of SRs is allocated to the schedule table after all other statements in the cache reference groups have been allocated, the latter statements including statements which would cause cache conflict with the removed statement and those statements in the cache group that have been allocated to the scheduling table.

After step **584** completes its execution, execution proceeds to step **586** which determines whether or not the set of SRs is empty. If the set of SRs is empty, i.e., all its statements have been allocated to the schedule table, then decision step **586** routes execution, via its true path, to step **590**. Otherwise, if the set of SRs is not empty, then decision step **586** routes execution, via its false path, to step **588**. Thus, one can see that step **586** effectively determines whether there are no statements which can be executed at time slot t other than memory reference statements which cause cache conflict. In other words, when the set of SRs is empty, there are no statements which can be executed at time slot t other than memory references which cause cache conflict, hence execution proceeds forward to step **590**. Alternatively, when the set of SRs is not empty, there are statements which can be executed at time slot t which do not cause cache conflict, thus effectively directing execution forward to step **588**.

As noted above, step **590** is executed when there are no statements which can be executed at time slot t other than memory references which cause cache conflict. When executed, step **590** resets the set of SRs to select a statement to be allocated to time slot t from among the memory reference statements which cause cache conflict. In other words, the set of SRs is reset to the contents of set SRorig, which is the contents of the set of SRs at the point this set was first fabricated through execution of step **578**. In this case, cache conflict is always generated, but as described above, the only statements which can be executed at this time, i.e., time slot t, are memory reference statements. Since it is rare to only have statements which cause cache conflict, this situation poses no great problem.

Alternatively, when such conflicting memory reference statements potentially executable at time slot t remain, then one can cease statement allocation at time slot t, increase the time slot value, t, by 1, and continue processing downward from step **578** in an attempt to allocate statements, including the conflicting memory reference statements, at time slot t+1. However, with this approach, unless one were to fix, i.e., limit, the number of time slots which are allowed to remain empty, i.e., have nothing allocated to them, processing may continue indefinitely. In view of the above caveat, this alternate approach can certainly be used within my inventive method, and particularly with step **570**.

In any event, step **588**, when executed, allocates one of the highest priority statements in the set of SRs to the time slot t portion of the schedule table. Once this occurs, execution loops back to step **578**. In particular, when step **588** is executed immediately after step **586**, then of the statements, which can be executed at time slot t, those that do not cause cache conflict and which have the highest priority are allocated to the schedule table. Alternatively, when step **588** is executed immediately after step **590**, then statements which cause cache conflict can be allocated to the schedule table, but only those statements having the highest priority are selected. There are various existing conventional methods for selecting one of many statements that has the highest priority which can be used in implementing step **588**. Since the particular method chosen is not particularly relevant to my present invention, I will omit any details of these methods herein.

Eventually, decision block **580** routes execution, over its true path, to decision step **592**. Step **592**, when executed, determines whether there are statements which have not yet been allocated to the schedule table. If such statements remain to be allocated, then decision step **592** routes execution, via its true path, to step **596**, which increments the current time slot value, t, by one. Thereafter, execution loops back to step **578** where processing is repeated for these statements. Alternatively, if all the statements in the intermediate code have been allocated, i.e., no unallocated statements remain, then processing of step **570** ends, with decision block **592** routing execution, via its false path, to point **594**, and from there to step **598**, which has been described above.

FIGS. **24**A and **24**B collectively show an illustrative schedule table **26** to which statements have been allocated by the above-described procedure and in accordance with data dependency graph **24** shown in FIG. **23**, the correct orientation of the drawing sheets for FIGS. **24**A and **24**B is shown in FIG. **24**. As one can now see, memory reference statements for the intermediate code and belonging to cache conflict groups, e.g., group **9010**, with cache reference group numbers **1** (stored within cache reference group number field **9020** shown in FIG. **18**), **2** and **5** which would otherwise cause cache conflict with each other, as illustrated in FIGS. **21**A and **21**B, through processing provided by, e.g., step **584**, are now sequentially lined up without being mixed together. Cache conflict groups **9010** with cache reference group numbers (**9020**) **3** and **4** are the same. From this, memory blocks are prevented from being rejected by cache, before all referencing to memory blocks previously transferred to a cache block, would otherwise have been completed.

FIGS. **25**A and **25**B collectively show intermediate code **5** after it has been reordered according to schedule table **26** shown in FIGS. **24**A and **24**B. In other words, intermediate code **5** shown in FIGS. **25**A and **25**B is intermediate code which resulted from use of cache conflict reduction process **20**, shown in FIG. **1**. After cache conflict reduction process **20**, code generation process **60** of compiler **1** is performed on the intermediate code shown in FIGS. **25**A and **25**B to generate object program **7**, generally shown in FIG. **1** and specifically shown collectively in FIGS. **26**A and **26**B; the proper orientation of the drawing sheets for FIGS. **26**A and **26**B, is depicted in FIG. **26**.

FIG. **27** shows the cache conflict for object program **7** shown in FIGS. **26**A and **26**B. As explained previously, FIG. **7** shows corresponding cache conflict when my inventive cache conflict reduction process **20** is not used on the same source program. As is clearly evident in FIG. **27**, one can see that, in contrast to that shown in FIG. **7**, there are absolutely no cache misses caused by cache conflicts.

I will now discuss the quantitative effects of this preferred embodiment. With computers currently available on the market, instruction execution time normally consumes one cycle per instruction, but when a cache miss occurs, such execution time can consume approximately **10** such instruction cycles. For the following discussion and for simplicity, I will estimate execution times using these values.

First, from the cache conflict generation conditions shown in FIG. **7**, when object program **5** shown in FIG. **6** was executed, without having previously performed my inventive cache conflict reduction method, an execution time of 4 loop cycles is obtained. For each of 4 iterations through the single loop, 6 cache misses are generated. Given my assumption of 10 cycles for each cache miss, and since the number of instructions for which a cache miss was not generated is 7, the total instruction execution time for the program shown in FIG. **6** is 4*(6*10 cycles+7 cycles)=268 cycles.

In contrast, from the cache conflict generation conditions shown in FIG. **27**, i.e., for object program **5** of FIGS. **26**A and **26**B which results from performing my inventive cache conflict reduction process on the same source program as that which resulted in the object code shown in FIG. **6**, given the same number of loop cycles, i.e., 4, the total execution time is the number of cache misses, here 6, multiplied by 10 cycles/miss plus time needed to execute the remaining instructions which did not generate a cache miss, i.e., 33 cycles. The resulting execution time is 6*10+33=93 cycles. Therefore, through use of my inventive cache conflict reduction process, the execution time, for the same source code, decreased from 268 to 93 cycles. In other words, execution time, as measured in cycles, decreased by a factor of 93/268 which equals an increase in execution speed of approximately 2.9 times.

Furthermore, by using the preferred embodiment to reorder the memory references in program intermediate code **5** through execution of memory reference reordering process **50**, any extra dependency caused by allocating a few registers to the memory reference instructions does not occur. Hence, use of my inventive method provides increased freedom to reorder memory references. Consequently, memory references can be readily transferred to optimal positions with a concomitant improvement in the amount of cache conflict reduction that results.

Further, with my preferred embodiment, memory reference reordering process **50** was described as being executed on program intermediate code **5**. However, process **50** can also be executed on object program **7**. In the latter case, instruction reordering for all instructions, including those which were not expected to be generated at the intermediate code stage, can be performed, which will likely produce an increasingly precise reordering.

Additionally, memory reference reordering process **50** can also be executed on both program intermediate code **5** and object program **7**. Since this case simultaneously provides all the advantages set forth in the preceding two paragraphs, optimal cache conflict reduction can be achieved.

Moreover, memory reference reordering process **50** can be executed on program intermediate code **5**, with the resulting intermediate code being reverse compiled (reconverted) to a corresponding source program, with this source program then being externally disseminated for use on computers other than that which generated this source program. Hence, through this approach, other compilers can also use the advantageous effects of cache conflict reduction process **20** which will have been incorporated into the source program, thereby enhancing the general use of cache conflict reduction process **20**.

I have shown and described the preferred embodiment of my inventive method, in the context of illustrative use with direct map method cache. However, those skilled in the art will realize that my inventive method is not limited to only direct map method cache, but rather can be used with other forms of cache, such as set associative method cache.

Through use of my invention, programs, which would otherwise exhibit a large decrease in performance due to cache conflicts, can generate fewer cache conflicts. As a result, program execution speed can be increased. My invention is very useful when compiling programs, operating on computers, which have direct map method cache.

Clearly, it should now be quite evident to those skilled in the art, that while my invention was shown and described, in detail, in the context of a preferred embodiment, and with various modifications thereto, a wide variety of other modifications can be made without departing from the scope of my inventive teachings.

I claim:

1. A compiling method for reducing cache conflicts which would be generated during the execution of a program, wherein the method compiles a source program of the program into an object program for a computer having a main memory and a data cache memory, the method comprising steps of:

a) analyzing syntax of said source program and generating initial intermediate code;

b) extracting memory references from said intermediate code;

c) generating a cache reference group for each group of said memory references that references a common memory block so as to form a plurality of cache reference groups;

d) classifying said cache reference groups and generating cache reference group information;

e) based on said cache reference group information, generating a cache conflict graph which shows the cache conflict conditions among said cache reference groups;

f) providing said cache reference group information and said cache conflict graph as said cache conflict information;

g) based on said cache conflict information, changing an execution order of memory reference codes in said initial intermediate code, so as to form reordered intermediate code which, when compiled into object code and ultimately executed, will generate fewer cache conflicts than would said initial intermediate code, if said initial intermediate code were to be compiled into object code and executed; and

h) generating an object program from said reordered intermediate code.

2. The method in claim **1** wherein the step (d) comprises the steps of:

determining, in response to whether a distance on said main memory between memory references in said initial intermediate code is less than a designated value, if said memory references refer to a common one of the memory blocks; and

registering those of said memory reference, which refer to a common one of the memory blocks, to a common one of the cache reference groups.

3. The method of claim **1** wherein the step (d) further comprises the steps of:

obtaining the memory block which references the positions, in said main memory, of memory references of said intermediate code, and

registering those of said memory references, which refer to the common memory block, to a corresponding common cache reference group.

4. The method of claim **1** wherein the step (e) comprises the steps of:

obtaining, for arbitrary first and second ones of the cache reference groups included in said cache reference group information, a minimum distance value in the cache memory between all of the memory references included in the first cache reference group and all of the memory references included in the second cache reference group; and

determining, in response to whether said minimum distance is less than the designated value, an occurrence of cache conflict between the memory references of said first cache reference group and the memory references of said second cache reference group.

5. The method claim **1** wherein the step (e), for arbitrary first and second cache reference groups included in said cache reference group information, comprises the steps of:

obtaining a first one of the cache blocks, said first cache block being mapped from positions in said main memory of all of the memory references that are included in said first cache reference group;

obtaining a second one of the cache blocks, said second cache block being mapped from positions in said main memory of all of the memory references that are included in said second cache reference group; and

determining, in response to whether ones of the memory reference are mapped to a same one of the cache blocks, existence of cache conflict between the memory references of said first cache reference group and the memory references of said second cache reference group.

6. A compiling method for reducing cache conflicts which would be generated during the execution of a program, wherein the method compiles a source program of the program into an object program for a computer having a main memory and a data cache memory, the method comprising steps of:

a) analyzing syntax of said source program and generating initial intermediate code;

b) detecting data cache conflicts which would be generated by memory reference codes included in said initial intermediate code and providing conditions for memory reference codes to generate cache conflicts as cache conflict information;

c) based on said cache conflict information, changing an execution order of memory reference codes in said initial intermediate code, so as to form reordered intermediate code which, when compiled into object code and ultimately executed, will generate fewer cache conflicts than would said initial intermediate code, if said initial intermediate code were to be compiled into object code and executed;

d) selecting the loop from one of a plurality of loops in said initial intermediate codes so as to identify a selected loop;

e) determining a number of times the selected loop is to be unrolled, such that a length of each memory reference that results once the selected loop is unrolled equals a block length of one of the cache blocks, and unrolling the selected loop said number of times; and

f) changing the order of said memory reference code in said intermediate code after the selected loop, which provides repetitive processing in said initial intermediate code, is unrolled.

* * * * *

US006539541B1

(12) **United States Patent**
Geva

(10) **Patent No.: US 6,539,541 B1**
(45) **Date of Patent: Mar. 25, 2003**

(54) METHOD OF CONSTRUCTING AND UNROLLING SPECULATIVELY COUNTED LOOPS

(75) Inventor: **Robert Y. Geva**, Cupertino, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/378,632**

(22) Filed: **Aug. 20, 1999**

(51) **Int. Cl.**^{7} **G06F 9/45**

(52) **U.S. Cl.** **717/150**; 717/151; 717/160; 712/241; 712/233; 712/239

(58) **Field of Search** 717/9, 5, 150, 717/151, 160; 712/216, 218, 233, 241, 239

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,361,354	A	*	11/1994	Greyzck	717/160
5,526,499	A		6/1996	Bernstein et al.	712/216
5,537,620	A	*	7/1996	Breternitz, Jr.	717/9
5,613,117	A	*	3/1997	Davidson et al.	717/144
5,664,193	A	*	9/1997	Tirumalai	717/5
5,724,536	A		3/1998	Abramson et al.	712/216
5,778,210	A		7/1998	Henstrom et al.	712/218
5,797,013	A	*	8/1998	Mahadevan et al.	717/9
5,802,337	A		9/1998	Fielden	712/216
5,809,308	A	*	9/1998	Tirumalai	717/9
5,835,776	A	*	11/1998	Tirumalai et al.	717/9
5,842,022	A	*	11/1998	Nakahira et al.	717/9
5,854,933	A	*	12/1998	Chang	717/9
5,854,934	A	*	12/1998	Hsu et al.	717/9
5,862,384	A		1/1999	Hirai	717/9
6,035,125	A	*	3/2000	Nguyen et al.	717/9
6,145,076	A	*	11/2000	Gabzdyl et al.	712/241
6,192,515	B1	*	2/2001	Doshi et al.	717/9
6,247,173	B1	*	6/2001	Subrahmanyam	717/9
6,263,427	B1	*	7/2001	Cummins et al.	712/236
6,263,489	B1	*	7/2001	Olsen et al.	717/129
6,269,440	B1	*	7/2001	Fernando et al.	709/106
6,289,443	B1	*	9/2001	Scales et al.	708/300
6,327,704	B1	*	12/2001	Mattson, Jr. et al.	717/153
6,343,375	B1	*	1/2002	Gupta et al.	717/152
6,367,071	B1	*	4/2002	Cao et al.	717/160
6,401,196	B1	*	6/2002	Lee et al.	711/213

OTHER PUBLICATIONS

TITLE: Improving Instruction Level Parallelism by loop unrolling dynamic memory disambiguation, ACM, Davidson et al, Dec. 1995.*
TITLE: Combining Loop Transformation considering caching and scheduling, ACM, author: Wolf et al, 1996.*
TITLE: Unrolling Loops with indeterminate loop counts in system level pipelines, Guo et al, IEEE, 1998.*
TITLE: Symbolic range propagation, author: Blume et al, IEEE, 1995.*
"Advanced Compiler Design & Implementation" by Steven S. Muchnick, cover, table of contents and pp. 547–569, Copyright 1997.

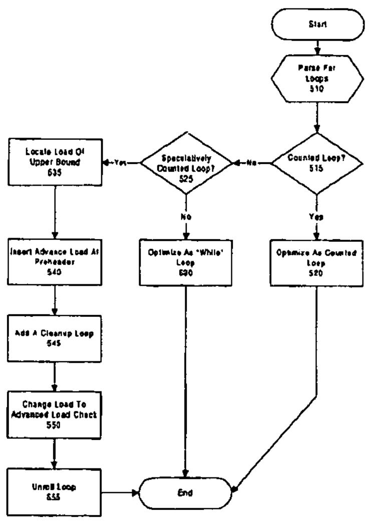
* cited by examiner

Primary Examiner—Kakali Chaki
Assistant Examiner—Chameli C. Das
(74) *Attorney, Agent, or Firm*—Peter Lam

(57) **ABSTRACT**

A method of constructing and unrolling speculatively counted loops. The method of the present invention first locates a memory load instruction within the loop body of a loop. An advance load instruction is inserted into the preheader of the loop. The memory load instruction is replaced with a check instruction. The loop body is unrolled. A cleanup block is generated for said loop.

32 Claims, 4 Drawing Sheets


Start
Parse For Loops 510
Counted Loop? 515
No
Speculatively Counted Loop? 525
Yes
Locate Load Of Upper Bound 535
Yes
Optimize As Counted Loop 520
No
Optimize As "While" Loop 530
Insert Advance Load At Preheader 540
Add A Cleanup Loop 545
Change Load To Advanced Load Check 550
Unroll Loop 555
End

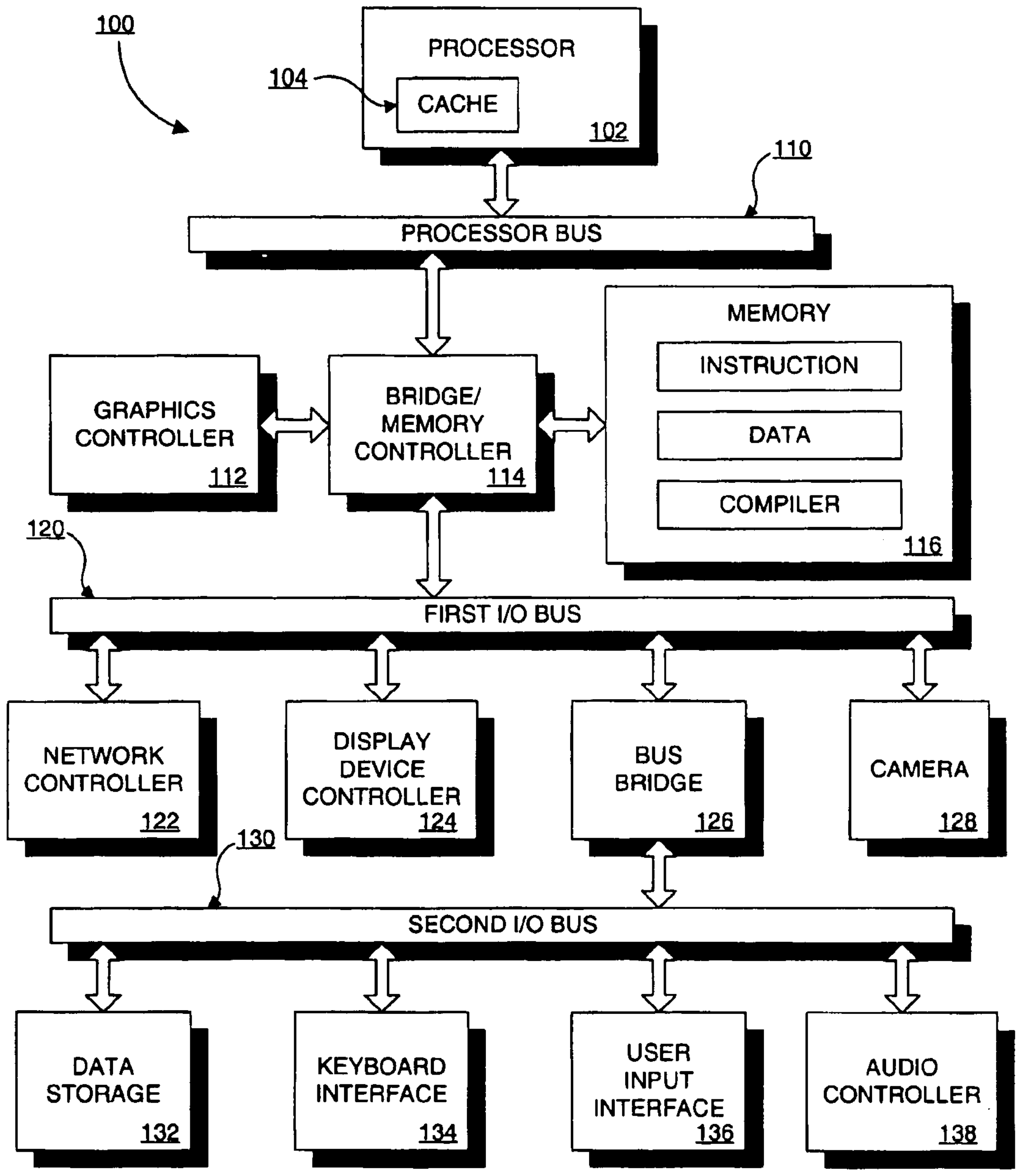

100
104
PROCESSOR
CACHE
102
110
PROCESSOR BUS
MEMORY
INSTRUCTION
DATA
COMPILER
116
GRAPHICS CONTROLLER
112
BRIDGE/ MEMORY CONTROLLER
114
120
FIRST I/O BUS
NETWORK CONTROLLER
122
DISPLAY DEVICE CONTROLLER
124
BUS BRIDGE
126
CAMERA
128
130
SECOND I/O BUS
DATA STORAGE
132
KEYBOARD INTERFACE
134
USER INPUT INTERFACE
136
AUDIO CONTROLLER
138

FIG. 1

```
for i := 1 to 100 do {
    s := s + a[i]
}
```

FIG. 2A

```
...
w = 4
store x = R2
t = q + s
z = x + w + u
R3 = load y
c = R3 + w
...
```

FIG. 3A

```
for i := 1 by 2 to 99 do begin {
    s := s + a[i]
    s := s + a[i + 1]
end}
```

FIG. 2B

```
...
R3 = ld.a y
w = 4
store x = R2
t = q + s
z = x + w + u
chk.a R3
c = R3 + w
...
```

FIG. 3B

```
i = 0
Loop:
    Body(i)
    i = i +1
    if (i < n) goto Loop
        else goto Exit
Exit:
```

FIG. 4A

```
i = 0
Loop:
    Body(i);
    i = i +1
    if (i >= n) goto Exit
    Body(i +1)
    i = i +1
    if (i >= n) goto Exit
    Body (i + 2)
    i = i +1
    if (i < n) goto Loop
        else goto Exit
Exit:
```

FIG. 4B

```
i = 0
t = ld.a n
if (t < 3) goto Recovery
Loop:
    Body(i)
    chk.a t, Recovery
    Body(i + 1)
    chk.a t, Recovery
    Body(i +2)
    i = i +3
    If (i < t-2) goto Loop
        else goto Recovery
Recovery:
    If (i >= n) goto Done
    Body(i)
    i = i +1
    If (i < n) goto Recovery
        else goto Done
Done:
```

FIG. 4C

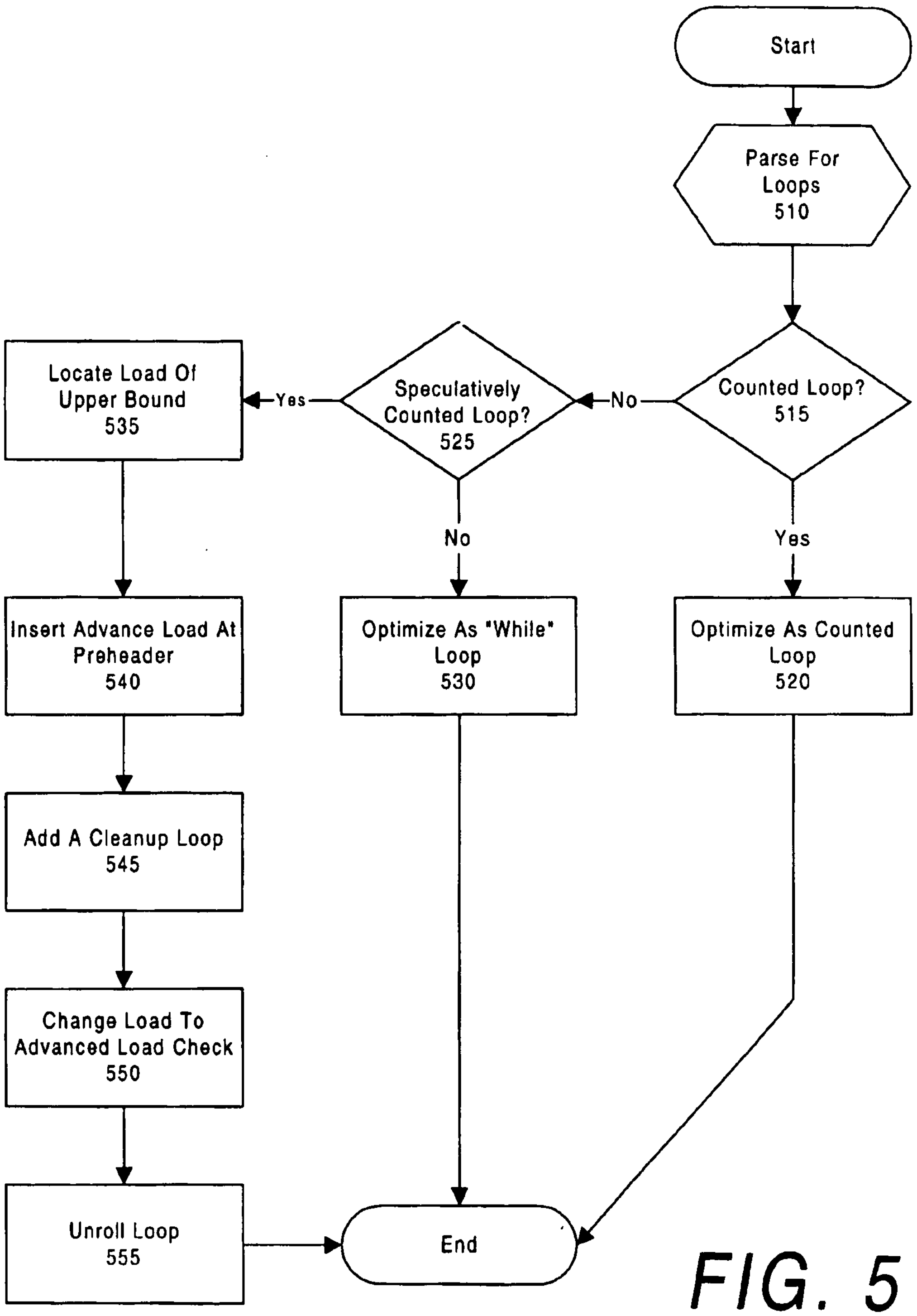

Start
Parse For Loops 510
Counted Loop? 515
No
Yes
Speculatively Counted Loop? 525
Yes
No
Locate Load Of Upper Bound 535
Insert Advance Load At Preheader 540
Add A Cleanup Loop 545
Change Load To Advanced Load Check 550
Unroll Loop 555
Optimize As "While" Loop 530
Optimize As Counted Loop 520
End

FIG. 5

METHOD OF CONSTRUCTING AND UNROLLING SPECULATIVELY COUNTED LOOPS

FIELD OF THE INVENTION

This invention relates to the field of computer software optimization. More particularly, the present invention relates to a method of constructing and unrolling speculatively counted loops. BACKGROUND OF THE INVENTION

Computer programs are generally created as source code using high-level languages such as C, C++, Java, FORTRAN, or PASCAL. However, computers are not able to directly understand such languages and the computer programs have to be translated or compiled into a machine language that a computer can understand. The step of translating or compiling source code into object code process is performed by a compiler. Optimizations are mechanisms that provide the compiler with equivalent ways of generating code. Even though optimizations are not necessary in order. for a compiler to generate code correctly, object code may be optimized to execute faster than code generated by straight forward compiling algorithm if code improving transformations are used during code compilation. Loop unrolling is one such optimization that can be used in a compiler.

For the purpose of loop unrolling, loops are categorized as follows. A loop is counted if the number of iterations that the loop will execute is determined once execution reaches the loop. Counted loops are also referred to as "for" loops. Conversely, a loop has data dependent exit, loosely called a "while" loop, if the number of iterations is determined during the execution of the loop. Counted loops are further classified. If the compiler can determine the number of iterations that the loop will execute at compile time, then the number of iterations is a compile time constant. Otherwise, the number of loop iterations is variable.

A compiler can optimize counted loops better than "while" loops. Applying the counted loop optimization to a "while" loop will cause the compiler to generate incorrect code. Therefore, in order to ensure the generation of correct code, the compiler's default assumption must be that all loops are "while" loops. Then the compiler may later try to prove that a loop is a counted loop so that more optimizations become possible. Similarly, the compiler can optimize a compile time constant counted loop to execute more efficiently than a variable counted loop. Furthermore, applying compile time constant loop optimizations to a variable counted loop will generate incorrect code. The compiler's default assumption has to be that all loops are variable, and only if the compiler succeeds in proving that a counted loop is a compile time constant counted loop, can the compiler proceed to apply further optimizations.

For example, here are two possible optimizations that a compiler can apply only to compile time constant counted loops. In one possible optimization, the compiler may unroll the loop entirely. Typically, compilers will unroll a loop entirely if the trip count is determined to be small, e.g. eight or less. A second optimization that a compiler may apply to loops with large compile time constant trip counts is to chose an unrolling factor that divide the trip count evenly. If the loop is variable or if such an optimal factor can not be found (e.g. if the trip count is a large prime number), then a cleanup loop must be generated after the unrolled loop to execute the remainder of the iterations.

Compilers can also optimize counted loop to execute more efficiently than "while" loops. In the context of loop unrolling, when the compiler unrolls a "while" loop, the

compiler has to simply copy the whole loop as many times as given by the unrolling factor chosen. This copy step includes the loop overhead. To illustrate, consider the following scheme:

```
LOOP:
  BODY(I)
  I=SOME_NEW_VALUE(I)
  If (CONDITION(I))
    GOTO LOOP
  ELSE
GOTO LOOP_END
LOOP_END:
```

BODY(I) is the useful part of the loop that does the real work in an iterative way. CONDITION is some test statement involving a variable "I" that changes in at least some of the loop iterations and that determines whether the loop terminates or continues execution. Unrolling this "while" loop by an unrolling factor of three yields the following construct:

```
LOOP:
  BODY(I)
  I=SOME_NEW_VALUE(I)
  If (NOT CONDITION(I))
    GOTO LOOP_EXIT
  BODY(I)
  I=SOME_NEW_VALUE(I)
  If (NOT CONDITION(I))
    GOTO LOOP_EXIT
  BODY(I)
  I=SOME_NEW_VALUE(I)
  IF (CONDITION(I))
    GOTO LOOP
  ELSE
    GOTO LOOP_EXIT
LOOP_EXIT:
```

When a compiler unrolls a counted loop, the compiler can save the loop overhead. The compiler can generate loop overhead code only once in each new iteration that corresponds to several original iterations. Consider the following counted loop construct:

```
I=0;
N=some_unknown_value;
LOOP:
  BODY(I)
  I=I+1
  If (I<N)
    GOTO LOOP
  ELSE
    GOTO LOOP_EXIT
LOOP_EXIT
```

Assume that the compiler decided to unroll this loop by an unrolling factor of three. The compiler has to generate code that will verify, at execution time, that the loop is about to execute at least three iterations. Also, the upper bound in the unrolled loop must now be reduced to N−2, and a cleanup loop must be generated to execute the remainder of the iterations. The resulting code will look like:

```
I=0
N=some_unknown_value
If (N<3) GOTO IN_BETWEEN
LOOP:
  BODY(I)
  BODY(I+1)
  BODY(I+2)
  If (I<N-2) GOTO LOOP
  ELSE GOTO IN_BETWEEN
```

```
IN_BETWEEN:
  IF (I>=N) GOTO LOOP_EXIT
CLEANUP:
  BODY(I)
  I=I+1
  IF(I<N)
    GOTO CLEANUP
  ELSE
    GOTO LOOP_EXIT
LOOP_EXIT
```

If the value of 'N' is large enough, most of the execution time will be spent in the unrolled loop. The added control around the loop has a negligible effect on performance. Significant performance is gained from not having to execute the loop overhead. Hence the compiler's ability to prove that a given loop is counted is a key in achieving this performance gain.

SUMMARY OF THE INVENTION

A method of constructing and unrolling speculatively counted loops is described. The method of the present invention first locates a memory load instruction within the loop body of a loop. An advance load instruction is inserted into the preheader of the: loop. The memory load instruction is replaced with an advanced load check instruction. The loop body is unrolled. A cleanup block is generated for said loop.

Other features and advantages of the present invention will be apparent from the accompanying drawings and from the detailed description that follow below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitations in the figures of the accompanying drawings, in which like references indicate similar elements, and in which:

FIG. 1 is a block diagram illustrating a computer system which may utilize the present invention.

FIG. 2A is an example 'for' loop before loop unrolling;

FIG. 2B shows the 'for' loop of FIG. 2A after the loop unrolling transformation;

FIG. 3A is a load-store pair in a code stream;

FIG. 3B shows the code of FIG. 3A after an advance load.;

FIG. 4A illustrates a loop before the loop unrolling transformation;

FIG. 4B illustrates the loop of FIG. 4A if unrolled by a factor of three as a 'while' loop;

FIG. 4C illustrates the loop of FIG. 4A unrolled three times as a speculatively counted loop; and

FIG. 5 is a flow diagram that illustrates steps for constructing and unrolling. speculatively counted loops in one embodiment of the present invention.

DETAILED DESCRIPTION

A method of constructing and unrolling speculatively counted loops is disclosed. Although the following embodiments are described with reference to C compilers, other embodiments are applicable to other types of programming languages that use compilers. The same techniques and teachings can easily be applied to other embodiments and other types of compiled object code.

The state of the art is proving that a loop is a counted loop includes the following steps. First, identify a variable, called

an induction variable, that changes in every loop iteration. The amount of change, called the stride, is usually required to be additive and the same for each loop iteration. Then that same variable has to be compared to some other value, the upper bound, in a manner that controls whether the loop will terminate or continue to execute more iterations. Also, the second value in the comparison must be loop invariant. The upper bound can either be a compile time constant or stored in a memory location that can not change during the execution of the loop.

Note that the trip count, i.e. the number of iterations that the loop execute each time is given by "trip count=(upper bound—lower bound)/stride". If the upper bound, lower bound, and stride are all compile time constants, then so is the trip count. In cases where the upper bound is stored in a variable, the compiler has to prove that the variable cannot change during the execution of the loop. In order to prove that, the compiler has to verify that each operation in the loop that changes a value stored in some memory location, such as a store operation, is targeting a memory location that is different from the one used to store the value of the loop upper bound. The process by which the compiler determines whether two memory access operation refer to overlapping areas in memory or not is called memory disambiguation, and is undecidable.

The enhancement disclosed here is a new way to use the data speculative loads, also known as advanced loads. The advanced loads are meant to help the compiler promote the location of a load instruction beyond store instructions that are not disambiguated. The new usage of advanced loads described in this invention is more powerful in that it allows the compiler to change the way it optimizes a whole loop rather than simply change the location of a single load instruction. The present invention enables a compiler to optimize these loops as speculatively counted. Optimizing certain loops as speculatively counted may allow code performance almost as good if the loops were optimized as counted loops and better than if the loops were optimized as while loops. Thus this invention may allow a compiler with such a capability to have a performance advantage over compilers that do not have this technology. As a result, it is important for the code optimizations to be effective. Therefore, a method of constructing and unrolling speculatively counted loops would be desirable.

Embodiments of the present invention may be implemented in hardware or software, or a combination of both. However, embodiments of the invention may be implemented as computer programs executing on programmable systems comprising at least one processor, a data storage system including volatile and non-volatile memory and/or storage elements, at least one input device, and at least one output device. Program code may be applied to input data to perform the functions described herein and generate output information. The output information may be applied to one or more output devices. For purposes of this application, a processing system includes any system that has a processor, such as, for example, a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The programs may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The programs may also be implemented in assembly or machine language. The invention is not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

The programs may be stored on a storage media or device (e.g., hard disk drive, floppy disk drive, read only memory

(ROM), CD-ROM device, flash memory device, digital versatile disk (DVD) or other storage device) readable by a general or special purpose programmable processing system, for configuring and operating the processing system when the storage media or device is read by the processing system to perform the procedures described herein. Embodiments of the invention may also be considered to be implemented as a machine readable storage medium, configured for use with a processing system, where the storage medium so configured causes the processing system to operate in a specific and predefined manner to perform the function described herein.

There are two possible computer systems of interest. The first system is called a "host". The host includes a compiler. The host system carries out the transformation of constructing and unrolling speculatively counted loops. The second system is called a "target". The target system executes the programs that were compiled by the host system. The host and target systems can have the same configuration in some embodiments. In the compiled program, speculatively counted loops can be present. Such a program would use the data speculation that is implemented in system hardware. The target computer system has to be one in which the processor has data speculation implemented.

An example of one such processing system is shown in FIG. 1. Sample system **100** may be used, for example, to execute the processing for embodiments of a method of constructing and unrolling speculatively counted loops, in accordance with the present invention, such as the embodiment described herein. Sample system **100** is representative of processing systems based on the PENTIUM®, PENTIUM® Pro, and PENTIUM® II microprocessors available from Intel Corporation, although other systems (including personal computers (PCs) having other microprocessors, engineering workstations, set-top boxes and the like) may also be used. In one embodiment, sample system **100** may be executing a version of the WINDOWS™ operating system available from Microsoft Corporation, although other operating systems and graphical user interfaces, for example, may also be used.

FIG. **1** is a block diagram of a system **100** of one embodiment of the present invention. System **100** can be a host or target machine. The computer system **100** includes a processor **102** that processes data signals. The processor **102** may be a complex instruction set computer (CISC) microprocessor, a reduced instruction set computing (RISC) microprocessor, a very long instruction word (VLIW) microprocessor, a processor implementing a combination of instruction sets, or other processor device, such as a digital signal processor, for example. FIG. **1** shows an example of an embodiment of the present invention implemented as a single processor system **100**. However, it is understood that embodiments of the present invention may alternatively be implemented as systems having multiple processors. Processor **102** may be coupled to a processor bus **110** that transmits data signals between processor **102** and other components in the system **100**.

System **100** includes a memory **116**. Memory **116** may be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, or other memory device. Memory **116** may store instructions and/or data represented by data signals that may be executed by processor **102**. The instructions and/or data may comprise code for performing any and/or all of the techniques of the present invention. A compiler for constructing and unrolling speculatively counted loops can be residing in memory **116** during code compilation. Memory **116** may also contain additional software and/or data not shown. A cache memory **104** may reside inside processor **102** that stores data signals stored in memory **116**. Cache memory **104** in this embodiment speeds up memory accesses by the processor by taking advantage of its locality of access. Alternatively, in another embodiment, the cache memory may reside external to the processor.

A bridge/memory controller **114** may be coupled to the processor bus **110** and memory **116**. The bridge/memory controller **114** directs data signals between processor **102**, memory **116**, and other components in the system **100** and bridges the data signals between processor bus **110**, memory **116**, and a first input/output (I/O) bus **120**. In some embodiments, the bridge/memory controller provides a graphics port for coupling to a graphics controller **112**. In this embodiment, graphics controller **112** interfaces to a display device for displaying images rendered or otherwise processed by the graphics controller **112** to a user. The display device may comprise a television set, a computer monitor, a flat panel display, or other suitable display device.

First I/O bus **120** may comprise a single bus or a combination of multiple buses. First I/O bus **120** provides communication links between components in system **100**. A network controller **122** may be coupled to the first I/O bus **120**. The network controller links system **100** to a network that may include a plurality of processing system and supports communication among various systems. The network of processing systems may comprise a local area network (LAN), a wide area network (WAN), the Internet, or other network. A compiler for constructing and unrolling speculatively counted loops can be transferred from one computer to another system through a network. Similarly, compiled code that has been optimized by a method of constructing and unrolling speculatively counted loops can be transferred from a host machine to a target machine. In some embodiments, a display device controller **124** may be coupled to the first I/O bus **120**. The display device controller **124** allows coupling of a display device to system **100** and acts as an interface between a display device and the system. The display device may comprise a television set, a computer monitor, a flat panel display, or other suitable display device. The display device receives data signals from processor **102** through display device controller **124** and displays information contained in the data signals to a user of system **100**.

In some embodiments, camera **128** may be coupled to the first I/O bus to capture live events. Camera **128** may comprise a digital video camera having internal digital video capture hardware that translates a captured image into digital graphical data. The camera may comprise an analog video camera having digital video capture hardware external to the video camera for digitizing a captured image. Alternatively, camera **128** may comprise a digital still camera or an analog still camera coupled to image capture hardware. A second I/O bus **130** may comprise a single bus or a combination of multiple buses. The second I/O bus **130** provides communication links between components in system **100**. A data storage device **132** may be coupled to second I/O bus **130**. The data storage device **132** may comprise a hard disk drive, a floppy disk drive, a CD-ROM device, a flash memory device, or. other mass storage device. Data storage device **132** may comprise one or a plurality of the described data storage devices. The data storage device **132** of a host machine can store a compiler for constructing and unrolling speculatively counted loops. Similarly, a target machine can store code that has been optimized by with a method for constructing and unrolling speculatively counted loops can be stored in data storage device **132**.

A keyboard interface **134** may be coupled to the second I/O bus **130**. Keyboard interface **134** may comprise a keyboard controller or other keyboard interface device. Keyboard interface **134** may comprise a dedicated device or may reside in another device such as a bus controller or other controller device. Keyboard interface **134** allows coupling of a keyboard to system **100** and transmits data signals from a keyboard to system **100**. A user input interface **136** may be couple to the second I/O bus **130**. The user input interface may be coupled to a user input device, such as a mouse, joystick, or trackball, for example, to provide input data to the computer system.

Audio controller **138** may be coupled to the second I/O bus **130**. Audio controller **138** operates to coordinate the recording and playback of audio signals. A bus bridge **126** operates to coordinate the recording and playback of audio signals. A bus bridge **126**. couples first I/O bus **120** to second I/O bus **130**. The bus bridge **126** operates to buffer and bridge data signals between the first I/O bus **120** and the second I/O bus **130**.

Embodiments of the present invention are related to the use of the system **100** for constructing and unrolling speculatively counted loops. According to one embodiment, such processing may be performed by the system **100** in response to processor **102** executing sequences of instructions in memory **116**. Such instructions may be read into memory **116** from another computer-readable medium, such as data storage device **132**, or from another source via the network controller **122**, for example. Execution of the sequences of instructions causes processor **102** to construct and unroll speculatively counted loops according to embodiments of the present invention. In an alternative embodiment, hardware circuitry may be used in place of or in combination with software instructions to implement embodiments of the present invention. Thus, the present invention is not limited to any specific combination of hardware circuitry and software.

The elements of system **100** perform their conventional functions well-known in the art. In particular, data storage device **132** may be used to provide long-term storage for the executable instructions and data structures for embodiments of methods of constructing and unrolling speculatively counted loops in accordance with the present invention, whereas memory **116** is used to store on a shorter term basis the executable instructions of embodiments of the methods of constructing and unrolling speculatively counted loops in accordance with the present invention during execution by processor **102**.

Although the above example describes the-distribution of computer code via a data storage device, program code may be distributed by way of other computer readable mediums. For instance, a computer program may be distributed through a computer readable medium such as a floppy disk, a CD ROM, a carrier wave, a network, or even a transmission over the internet. Software code compilers often use optimizations during the code compilation process in an attempt to generate faster and better code. Loop unrolling is one optimization that may be applied when code is compiled. An example of a typical loop may be:

```
Loop{. . .
  B(i)
  i=i+1
  test(i)
  exit
  . . . }
```

There is normally some control overhead such as 'test(i)' in the above example to control the number of loop iterations.

In loop unrolling, the loop body is copied multiple times. The above loop may be unrolled to become:

```
Loop{. . .
  B(i)
  B(i+1)
  B(i+2)
  B(i+3)
  i=i+4
  test (i)
  exit . . . }
```

The original loop body B(i) has been copied three times and the control variable 'i' incremented accordingly. By unrolling the loop, the branch instruction and test for loop exit executes three times less than in the original loop. Furthermore, fewer instructions are needed for the control flow and more instructions are grouped together into a block. A large contiguous block of code may also allow for subsequent code optimizations.

Loop unrolling reduces the overhead of executing an indexed loop and may improve the effectiveness of other optimizations, such as common subexpression elimination, induction-variable optimizations, instruction scheduling, and software pipelining. Loop unrolling generally increases the available instruction-level parallelism, especially if several other transformations are performed on the copies of the loop body to remove unnecessary dependencies. Thus, unrolling has the potential of significant benefit for many implementations and particularly for superscalar and VLIW ones and Explicitly Parallel Instruction Computing (EPIC).

Loop unrolling may also provide other advantages. For instance, instruction scheduling or prefetching in some computer architectures may benefit from loop unrolling. Loop unrolling is often used to enable the generation of data prefetch instructions. When a compiler inserts. data prefetch instructions into loops, the compiler may need to insert those instructions into only some iterations of the loop. Unrolling the loop makes several iterations explicitly available to the compiler such that the compiler can insert instructions to some and not all of iterations. In some instances, unrolled loops may utilize cache memory more efficiently. Furthermore, not taken branches may be cheaper in terms of performance loss than taken branches. If the compiler predicts that the loop will execute many iterations, then a larger block of code may be cached in memory and fewer jumps or branches will be executed.

When the compiler is analyzing the program code, a loop may be completely removed and replaced with a contiguous block of code if the number of loop iterations is small. Similarly, the number of loop iterations may be reduced through loop unrolling if the number of iterations is large.

Referring now to FIGS. **2A** and **2B**, there are two examples of 'for' loops. FIG. **2A** illustrates a normal 'for' loop before unrolling. FIG. **2B** illustrates the 'for' loop of FIG. **2A** after the loop unrolling transformation. The unrolling transformation in this example has been oversimplified. For this example, the loop bounds are known constants and the unrolling factor divides evenly into the number of iterations. However, such conditions are generally not satisfied and the compiler has to keep a cleanup copy of the loop. When the number of iterations remaining in a loop is less than the unrolling factor, the unrolled copy is exited and the cleanup copy is executed to complete the remaining iterations. This approach also reduces the number of early termination tests and conditional control flow between copies of the body in some loops.

When the compiler unrolled the loop of FIG. **2A** by a factor of two, the loop body "s:=s+a[i]" was copied twice and the loop counter 'i' adjusted as shown in FIG. 2B. The unrolled loop executes both the loop-closing test and the branch half as many times as the original loop. Hence loop unrolling optimization may positively affect system performance. In the present example, loop unrolling increases the effectiveness of instruction scheduling by making two loads of 'a[i]' values available to be scheduled in each loop iteration.

Loops are generally distinguished into two classifications: counted loops and loops with a data dependent exit. A loop is counted if the number of iterations that the loop will execute is determined once execution reaches the loop. However, if the number of loop iterations is not determined once execution reaches the loop, and is determined during the execution of the loop, then the loop will be classified as a while loop. The number of iterations that a while loop executes may be determined during the execution of the loop or on the fly.

Counted loops are further distinguished between two kinds. The first kind of loop has a constant number of iterations known at compile time. For example, the header of a loop may be "for (i=0; i<200; i++)". The compiler will be able to determine that the number of loop iterations will be two hundred. The loop may be unrolled, but the loop body may not necessarily be copied two hundred times. Instead, the unrolling factor may be a smaller number that divides into two hundred. The loop body may be copied ten times and the new loop executed twenty times. In one embodiment, the loop unrolling factor is chosen such that the loop count is divided evenly.

The second type of counted loop is variable. The point of a variable counted loop is that the compiler cannot determine the number of loop iterations. The inability to determine the number of iterations can be due to a variety of reasons. One example of the inability of a compiler to determine the number of iterations in compile time is the case where the value of loop iterations is read by the program from an input file. A function call is typically such a barrier to analysis. Compilers do perform inter function analysis. Conversely, a function call is not the only reason why a compiler is unable to figure out the number of loop iterations.

In a variable counted loop, the number of loop iterations can be the result of a function call. The function call provides a number to be used as the loop upper bound. As a result, the compiler will not know at compile time how many times the loop will execute since the loop count may be different each time the loop is executed. So even though the loop is a counted loop, the trip count is unknown or not a compile time constant.

In addition to counted loops and "while" loops, a third class of loops is introduced. This third class comprises speculatively counted loops. A speculatively counted loop satisfies all the requirements of a counted loop except for the characteristic that a speculatively counted loop has a loop upper bound that has not been proven to be loop invariant. Without the ability to classify loops as speculatively counted, these loops would have to be considered "while" loops. The compiler can transform a "while" loop into a speculatively counted loop by: (1) inserting an advanced load of the upper bound into a register; and (2) inserting an advanced load check before the loop termination test. Various optimizations such as software pipelining, whose effectiveness depends on classification of loops, may benefit from being able to transform "while" loops into speculatively counted loops, the following embodiments only demonstrate the way the loop unrolling optimization benefits from this

capability. The description of speculatively counted loops and methods of constructing speculatively counted loops are presented within the context of loop unrolling.

Knowledge that a loop is a counted loop allows the compiler the opportunity to further is optimize the loop in ways that may not be available otherwise. One such optimization may be loop unrolling where a loop is unrolled 'n' times, such that 'n−1' additional copies of the loop body are made. When the unrolled loop is a counted loop, there is no need to test for the exit condition inside the unrolled body. But if the loop is a data dependent or while loop, then the exit condition needs be tested after each loop body. Because the exit condition is tested once during each iteration of the counted loop, 'n−1' tests are saved per each iteration.

In order to classify a loop as a counted loop, the compiler has to prove a number of conditions. These conditions may include identifying characteristics such as a linear induction variable and a loop invariant variable that serves as an upper bound for the loop in particular. If the compiler cannot prove that the upper bound is loop invariant, then the loop cannot be classified as a counted loop. One of the most common limitations to proving that a variable is loop invariant is showing that no memory location stores that executes inside the loop body can change the value of the loop upper bound.

Unrolling a data dependent loop such as the following while loop may be generate less efficient code. while (a[i] !=0) do {. . . B(i) . . . }
The compiler first copies the loop body a number of times. In this example, the compiler is designed to unroll all loops by a factor of four. Next, the compiler has to insert multiple exit tests to check for termination between every loop body. Similarly, if the loop had a variant upper bound, test statements would be needed to ensure that the upper bound had not changed.

```
Loop {. . .
    B(i)
if (a[i]=0) goto Exit
    B(i+1)
if (a[i+1]=0) goto Exit
    B(i+2)
if (a[i+2]=0) goto Exit
    B(i+3)
if (a[i+3]=0) goto Exit
    else goto Loop
Exit . . . }
```

On the other hand, some counted loops may need the exit condition tested only once. In a counted loop, the original loop body is simply replaced with four copies of the loop body.

But unrolling a loop having an indeterminate number of iterations is more complicated. The compiler may attempt to unroll the loop even though the value of the loop count 'n' is unknown. If 'n' turns out to be two and the compiler had copied the loop four times, then the program code will be wrong since the loop will be executed four times before the exit condition is tested. Another issue in loop unrolling is that the trip count may not be evenly divisible. For instance, there may be no way to evenly divide a trip count of seventeen or nineteen. As a result, a clean up loop simply comprising the original loop with one loop body may be inserted after the unrolled loop. In the example loop having a trip count of seventeen, the processor may execute the unrolled loop with the four copies four times and the cleanup loop once for a total of seventeen iterations.

Some important issues in loop unrolling are deciding which loops to unroll and by what to factor. The concerns involved are architectural characteristics and the selection of particular loops in a particular program to unroll and the unrolling factors to use for them. Architectural characteristics include factors such as the number of registers available, the available overlap among floating-point operations and memory references, and the size and organization of the instruction cache. The impact of some architectural characteristics is often determined heuristically by experimentation. As a result, unrolling decisions for individual loops can benefit significantly based on feedback from profiled runs of the program.

The results of such experimentation may also depend on the presence of the following loop characteristics: (1) the presence of only a single basic block or straight-line code; (2) a balance of floating-point and memory operations or a certain balance of integer memory operations; (3) small number of intermediate-code instructions; and (4) loops having simple loop control. The first and second criteria restrict loop unrolling to loops that are most likely to benefit from instruction scheduling. The third characteristic attempts to keep the unrolled blocks of code short so that cache performance is not adversely impacted. The last criterion keeps the compiler from unrolling loops for which it is difficult to determine when to take the early exit to the unrolled copy for the final iterations, such as when traversing a linked list. In one embodiment of the invention, the unrolling factor may be anywhere from two on up, depending on the specific contents of the loop body. Furthermore, the unrolling factor of one embodiment will usually not be more than four and almost never more than eight. However, further development of VLIW or EPIC machines may provide good use for larger unrolling factors.

In one embodiment, the number of copies made of the loop body is determined heuristically. In another embodiment, the compiler may provide the programmer with a compiler option to specify which loops to unroll and what factors to unroll them by. A performance tradeoff exists depending on how many times the loop body is copied. One factor involved is the size of the instruction cache. Code performance may be impacted if a loop body is copied too many times since the block of new code may not fit into the instruction cache. A programmer may want to grow the number of instructions in a loop body so that the computer has a larger contiguous block of code to execute. However, the body of instructions should fit into the instruction cache or else a performance hit may occur. Hence, the programmer may start initially with a loop that originally fits in the instruction cache, but end up with a large block of instructions that no longer fits into the cache.

In the present invention, a new classification of loops called speculatively counted loops is created. Speculatively counted loops have generally been classified as data dependent exit loops and hence, not optimized as a counted loop. Speculatively counted loops have a construct similar to that of counted loops, but some speculatively counted loops may have stores to memory that cannot be disambiguated from the loop upper bound. Hence, the reason the compiler did not classify the loop as a counted loop was because the loop upper bound could not be disambiguated. One example where the compiler cannot prove that the upper bound is loop invariant is in a loop involving pointers to arrays. A speculatively counted loop would have been classified as a counted loop if the loop upper bound had been disambiguated from all memory stores in the loop. Hence, the reason the compiler did not classify the loop as a counted loop was because the loop upper bound could not be disambiguated. In one embodiment, the process of classifying a loop as

speculatively counted is performed in a procedure that is similar to the process used to classify loops as a counted loop.

Another problem encountered with loop unrolling is that the value of the trip count 'n' cannot change within the loop. Proving that the trip count is constant may be a difficult task for some compilers. For example, the program may have a pointer that points to an integer value. Depending on the program language, a pointer may generally be assigned a value anywhere within the program, including somewhere inside a loop. Furthermore, pointers may be dynamic and array lengths may change. If the compiler is unable to prove that none of the memory stores inside the loop change the value of 'n,' then the processor may not execute four iterations of the loop body consecutively without testing for loop termination between each body. For example, a loop may look like:

```
n=10
p=address n
for (i=0; i<n; i++) {. . .
  x=y+z
  *p=4
  . . . }
```

The header of the above loop is "for (i=0; i<n; i++)," where the loop count 'n' may be a variable dynamically defined by a function call or 'n' may be referenced by a pointer '*p' or modified within the loop body. The ambiguity introduced by pointer *p prevents loop optimization in conventional compilers. Since the upper bound 'n' is not known at compile time or may change within the loop, the compiler will consider the loop as having an unknown upper bound. Hence the loop would be treated like a while loop. The present invention may allow for the transformation of loops that look like counted loops, but have loop upper bounds that cannot be proven as loop invariant.

A statement in a computer program is said to define a variable if it assigns, or may assign. a value to that variable. For example, the statement "x=y+z" is said to define 'x'. A statement that defines a variable contains a definition of that variable. In this context there are two types of variable definitions: unambiguous definitions and ambiguous definitions. Ambiguous definitions may also be called complex definitions. When a definition always defines the same variable, the definition is said to be an unambiguous definition of that variable. For example, the statement, "x=y" always assigns the value of 'y' to 'x'. Such a statement always defines the variable 'x' with the value of 'y'. Thus, the statement "x=y" is an unambiguous definition of 'x'. If all definitions of a variable in a particular segment of code are unambiguous definitions, then the variable is known as an unambiguous variable.

Some definitions do not always define the same variable and may possibly define different variable at different times in a computer program. Thus they are called ambiguous definitions. There are many types of ambiguous definitions. One type of ambiguous definition occurs where a pointer refers to a variable. For example, the statement "*p=y" may be a definition of 'x' since it is possible that the pointer 'p' points to 'x'. Hence, the above ambiguous definition may ambiguously define any variable 'x' if it is possible that 'p' points to 'x'. In other words, '*p' may define one of several variables depending on the addressed value of 'p'. Another type of ambiguous definition is a call of a procedure with a variable passed by reference. When a variable is passed by reference, the address of the variable is passed to the procedure. Passing a variable by reference to a procedure allows the procedure to modify the variable. Alternatively, variables may be passed by value. Only the value of the variable is passed to the procedure when a variable is passed by value. Passing a variable by value does not allow the procedure to modify the variable. Another type of ambiguous definition is a procedure that may access a variable because that variable is within the scope of the procedure. Yet another type of ambiguous definition occurs when a variable is not within the scope of a procedure but the variable has been identified with another variable that is passed as a parameter or is within the scope of the procedure.

When the compiler unrolls a loop having a data dependent exit, the compiler makes copies of the loop body 'i' and the exit test. The exit test allows the processor to take side exits out of the loop during program execution. Data dependent exit loops are generally tested for loop termination between each copy of the body. If the loop has to terminate, then the processor has to go to a loop exit. If the exit condition tests true, then the loop has to terminate and the program goes to a loop exit. If the condition is false, then the next loop body 'i+1' is executed and the test for loop termination performed again.

One advantage of the present invention may be the omission of the exit condition test between copies of the loop body. However, the compiler needs to determine whether the speculatively counted loop may be correctly treated as a counted loop. If the compiler cannot make such a determination, then the tests for loop termination and side exits are kept in the loop. One criteria in determining if a speculatively counted loop may be treated like a counted loop is whether the loop upper bound is loop invariant. In order to prove that the upper bound is truly loop invariant, the compiler needs to analyze the stores that occur inside the loop. The process of proving that two memory operands are different is called memory disambiguation.

Data speculation occurs when a later load is scheduled above an earlier store and the compiler cannot verify that the load and store will never access overlapping areas of memory. The process of determining whether loads and stores access overlapping areas of memory is termed "disambiguation." A load-store pair for which the compiler cannot guarantee that the load and store will never access overlapping areas of memory are termed "un-disambiguated." In the following text, the phrase "un-disambiguated store" will be used to refer to the store in an un-disambiguated load-store pair. A store cannot be un-disambiguated by itself, but only in the context of a particular load.

Compilers often perform memory disambiguation to prove that a loop upper bound is loop invariant. Sometimes, two memory operands may appear to be different, but the compiler is unable to verify that the two operands are indeed different. Memory disambiguation attempts to verify that two variables are not the same and are not affected by changes to the other. In one embodiment, the processor may include a special construct to assist compilers in the task of memory disambiguation. One special construct for memory disambiguation is the advance load or data speculative load. For example, a program has stored a piece of data at memory location X. At some later point in the program, a piece of data is loaded from memory location Y. If the compiler tries to schedule the memory load before the memory store, the resulting program is legal only if locations X and Y are different memory locations. If the compiler can prove that memory locations X and Y are indeed different, then the compiler can switch the order of the store and load instructions. But if locations X and Y are the same memory location and the order of the store and load instructions are switched,

then the memory load would be fetching the wrong data since the correct data has not yet been stored at the memory location. If the variable that stores the loop upper bound cannot be disambiguated from all the memory stores in the loop, then that value has to be read/reloaded from memory prior to each comparison, ad the loop cannot be treated as a counted loop.

The compiler may use the advance load construct in situations where the compiler cannot verify that the memory locations are different. The present invention may be used with counted loops that have upper bounds that cannot be disambiguated from memory stores within the loop body. One example may be a loop that contains a pointer into a large array. The compiler may not be able to verify that the pointer does change loop upper bound. In one embodiment, the advance load (Id.a) and advance load check (chk.a) instructions interact with a hardware structure called the advanced load address table (ALAT). The advanced load instruction causes the processor to perform a load from a memory location and write the memory address into an ALAT. The ALAT acts as a cache of the physical memory address and the physical register address accessed by the most recently executed advanced loads. The size and configuration of the ALAT is implementation dependent. A straightforward implementation of one embodiment may have entries containing a physical memory address field, an access size field, a register address field, and a register type field (general or floating-point). Using the target register address and type as an index, advanced loads allocate a new entry in the ALAT containing the physical address and size of the region of memory being read.

During each memory store, the processor scans the ALAT for any entries having the same memory address. Store instructions would cause the processor to search all entries in the ALAT using the physical address and size of the region of memory being written. All entries corresponding to overlapping regions of memory are invalidated. Advanced load checks access the ALAT using the target register address and type as an index. If the corresponding ALAT entry is not valid, then either a store subsequent to the advanced load accessed an overlapping area of memory or the advanced load's entry has been replaced. The advanced load check then performs the normal load operation for memory access corresponding to the invalid ALAT entry. But if the ALAT entry accessed by the advanced load check is valid, then the advanced load had received correct data and the advanced load check performs no action.

One embodiment uses "advanced loads" or "data speculative loads" to handle un-disambiguated memory load-store pairs. Support for data speculation may take the form of the advance load (Id.a) and advance load check (chk.a) instructions. A memory load that is statically scheduled above an earlier store when the pair are un-disambiguated is converted into an advanced load. However, if the load-store pair can be disambiguated then the load does not need to be converted into an advanced load. When the compiler converts a particular load into an advanced load, a corresponding advanced load check is scheduled at a point below the lowest un-disambiguated store in the originating basic block of the advanced load. Thus the advanced load and advanced load check instructions bracket one or more un-disambiguated stores. The advanced load check should be configured to perform the same memory access in both address and size, and write the same destination register as the advanced load.

The advance load check constructs is related to the advance load. In one embodiment, the compiler will insert an advanced load check instruction between copies of the loop body in the unrolled loop. The advanced load check statement may be inserted just prior to the statement that uses the advance loaded data. The advanced load check instruction directs the processor to check the ALAT for a specific memory address. The advanced load check instruction checks to see if the advance loaded data has been modified by a memory store. If the data has been changed, then the data has to be reloaded. In one embodiment, a failed check indicates that the data advance loaded from the given memory location has been superseded with more recently stored data. If the desired memory address is missing from the ALAT or if the entry has been invalidated, then the check has failed and a memory load needs to be execute again for the specified memory location. In either situations of the missing ALAT entry or invalidated ALAT entry, a new memory load is performed so that the instruction requesting the desired data will be using the correct result. Hence by using the advanced load and advanced load check constructs in a program, the compiler can change the order of the loads and stores without causing the program to function incorrectly.

Referring to FIGS. 3A and 3B, use of the advance load and advanced load check are illustrated. FIG. 3A illustrates a load-store pair in a code stream. The "store x=R2" instruction represents a memory store of the contents of register 'R2' to memory operand 'x'. The "R3 load y" instruction represents a memory load of memory operand 'y' to register 'R3'. FIG. 3B illustrates the code after an advance load. In FIG. 3A, the "R3=load y" instruction may be moved above the "store x=R2" only if memory operands 'y' and 'x' are different. Otherwise, the move would be illegal. The advanced load check (chk.a) is used when a memory load is move earlier in the instruction stream for advance loading. The memory load of FIG. 3A has been moved earlier in the instruction stream and modified to become an advance load as illustrated by "R3=ld.a y" in FIG. 3B. Correspondingly, a advanced load check "chk.a R3" has been inserted at the original location of the memory load and just prior the use of register 'R3'. If the advance loaded value of register 'R3' from memory operand 'y' has been modified before the advanced load check, then memory load needs to occur again in order to correct the changes. If instructions that are data dependent upon the advanced load are not scheduled above an un-disambiguated store, then only the memory load instruction needs to be re-executed in the event of an overlap between the advanced load and a memory store. This operation is the function of the advanced load check. However, if one or more instructions dependent upon the advance load are scheduled above an un-disambiguated store, then in the event of an overlap all of these rescheduled instructions need to be re-executed in addition to the memory load.

The chk.a instruction is used to determine whether certain instructions needed to be re-executed. The compiler can use the advance load check (chk.a) if other instructions are also moved before the memory store. The advance load check branches the execution to another address for recovery if the check fails. The advance load check (chk.a) instruction of one embodiment has two operands. One operand is the register containing the data loaded by advance load. The second operand is the address of the recovery block. The recovery block can be simple and just branch to the cleanup loop in one embodiment. If the chk.a cannot find a valid ALAT entry for the advance load, then the program branches to a recovery routine in an attempt to fix any mistakes made by using the wrongly loaded data. The chk.a instruction

specifies a target register that needs to have the same address and type as the corresponding advanced load. In the event of an invalid entry in the ALAT, program control is transferred to a recovery block. The recover block contains code that comprises a copy of the advanced load in non-speculative form and all of the dependent instructions prior to the chk.a. After completion of the recovery code, the program resumes normal execution. However, the point at which normal execution is not predefined. The recovery block has to end with a branch instruction to redirect execution at a continuation point in the main thread of execution. One goal of the recovery block is to maintain program correctness. If a memory store in the loop body is changing the value of the upper bound, the recovery block or cleanup loop may also revert the loop back to its original form and simply iterates one loop body at a time.

The present invention discloses a method to optimize a speculatively counted loop. Unrolling speculatively counted loops is similar to unrolling counted and while loops. When a speculatively counted loop is unrolled, the loop body is copied 'n−1' times. The compiler also adds a statement into the preheader of the loop to perform a data speculative load of the loop upper bound from memory into a register. In another embodiment, the statement may be inserted at a point that is outside of the loop. The data speculative load is also referred to as an advanced load. Then between every two loop bodies in the unrolled loop, the compiler inserts a speculation check instruction. The check instruction of one embodiment is an advance load check. The speculation check is related to the advanced load that was added to the preheader. A speculation check determines whether the memory location that was speculatively loaded has been changed by a subsequent store to memory. If the speculatively loaded memory located has been changed, then control is transferred to the recovery block.

FIGS. 4A, 4B, and 4C illustrate three different versions of a loop. FIG. 4A illustrates the loop before the loop unrolling transformation. FIG. 4B illustrates the loop of FIG. 4A if unrolled by a factor of three as a 'while' loop. FIG. 4C illustrates the loop of FIG. 4A unrolled three times as a speculatively counted loop. The loop counter or control of all three versions is represented by 'i' and the termination count is represented by 'n'.

In one embodiment, the compiler may use the advance load and advanced load check constructs in a program loop if the only instruction relevant to the contents of a memory address moved before the memory store is the memory load. The compiler starts by generating an advance load of the upper bound. The loop body may then be copied and the count incremented accordingly. But instead of testing between each loop body for loop termination as in a while loop, the compiler generate an advanced load check that corresponds to the target of the advance load. The compiler also appends a cleanup loop having a single loop body to the unrolled loop. A failed check would cause a recovery and memory load to be performed so that the program execution could continue correctly. Furthermore, the compiler may also take certain instructions that use the value that was advance loaded, such as an add instruction, and move those instructions before the memory store in the code. The method of constructing and unrolling speculatively counted loops does not have to keep track of any specific store that cannot be disambiguated from the load of an upper bound. Once the load of the upper bound is not proven to to be loop invariant, there is no longer a need to keep track of a specific store. There may be any number of such stores. However, if the advanced load check fails, then the moved instructions may have to be re-executed again after the correct data is loaded in order to maintain program correctness.

The function of the advanced load check in one embodiment includes branching to a recovery block if the check fails. During the loop unrolling transformation of one embodiment, the compiler can generate code for the recovery block that will re-execute all the instructions that were moved in front of the memory store. The recovery block of one embodiment may also branch to a cleanup loop. Once the processor completes the recovery, the program may direct the processor to branch from the end of the recovery block to back a point in the program after the originating advanced load check. The processor can then continue program execution as before the check failed. Hence if the advanced load check does not fail, the overhead is negligible and the program may execute quickly. The compiler can generate a recovery block by saving a copy of the original loop. The recovery block contains code to perform a new load of the memory location into a register and to transfer loop control to a version of the loop that is identical to the original version of the loop. The speculation check instruction and the recovery block are measures to ensure correct loop execution. In one embodiment, the recovery block is not part of the actual loop and the check instruction is comprised of one instruction. Hence, the performance of an unrolled speculatively counted loop may approach that of code generated for an unrolled counted loop.

FIG. **5** is a flow diagram that illustrates steps for constructing and unrolling speculatively counted loops in one embodiment of the present invention. Software developers may often decide to optimize computer programs in attempt to improve performance. One such code optimization method may entail the steps as shown in FIG. **5**. The compiler parses the program code for loops at step **510**. When a loop is encountered at step **515**, the compiler determines whether the loop is a counted loop. If the loop is a counted loop, then the compiler attempts to optimize the loop as a counted loop at step **520**. If the loop is found not to be a counted loop, the compiler goes on to step **525** to determine whether the loop is a speculatively counted loop. If the loop is found not to be a speculatively counted loop, the compiler attempts to optimize the loop as a non-speculatively counted or "while" loop at step **530**. When the compiler has determined that a speculatively counted loop is present, load instructions of upper bounds are located within the loop at step **535**. Advance loads are inserted at the loop preheader at step **540**. At step **545**, the compiler generates and adds a cleanup loop. The cleanup block and recovery block in one embodiment may be identical or simply point to the other block of code. Memory load instructions are changed to advanced load check instructions at step **550**. The original loop body is unrolled at step **555**. The unrolling factor of one embodiment is determined heuristically. In another embodiment, the unrolling factor may be user specified or predetermined.

In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. For purposes of explanation, specific numbers, systems and configurations were set forth in order to provide a thorough understanding of the present invention. It will, however, be evident that various modifications and changes may be made thereof without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method of constructing an unrolled loop comprising:

identifying a speculatively counted loop, wherein said speculatively counted loop includes a loop upper bound that has not been proven to be loop invariant;

locating a memory load instruction within loop body of said speculatively counted loop;

inserting an advance load instruction into a preheader of said speculatively counted loop;

replacing said memory load with an advanced load check instruction; unrolling said loop body of said speculatively counted loop; and

generating a cleanup block for said speculatively counted loop.

2. The method of claim **1** further comprising converting a while loop into a speculatively counted loop.

3. The method of claim **1** wherein said loop body is unrolled by a predetermined unrolling factor.

4. The method of claim **1** further comprising moving instructions located within said loop from a first location after a memory store instruction to a second location before said memory store instruction in said loop.

5. The method of claim **1** wherein said cleanup block comprises a rolled copy of original loop body.

6. The method of claim **1** further comprising removing termination tests from between unrolled loop bodies.

7. The method of claim **1** further comprising generating a recovery block for said loop.

8. The method of claim **1** wherein said cleanup block is a recovery block.

9. A method of optimizing program performance comprising:

identifying a loop, said loop having a memory. load that cannot be disambiguated from a loop upper bound wherein said loop upper bound has not been proven to be loop invariant;

locating a memory load instruction for said memory load within loop body of said loop;

inserting an advance load instruction in preheader of said loop;

replacing said memory load instruction with an advanced load check instruction;

unrolling said loop body; and

generating a cleanup block.

10. The method of claim **9** wherein said loop is a speculatively counted loop.

11. The method of claim **9** wherein said loop is a data dependent while loop.

12. The method of claim **9** wherein said loop body is unrolled by a predetermined unrolling factor.

13. The method of claim **9** further comprising converting a while loop into a speculatively counted loop.

14. The method of claim **9** further comprising moving instructions located within said loop from a first location after a memory store instruction to a second location before said memory store instruction in said loop.

15. The method of claim **9** wherein said cleanup block comprises a rolled copy of original loop body.

16. The method of claim **9** further comprising removing termination tests from between unrolled loop bodies.

17. The method of claim **9** further comprising generating a recovery block for said loop.

18. The method of claim **9** wherein said cleanup block is a recovery block.

19. A computer readable medium having embodied thereon a computer program, the computer program being executable by a machine to perform:

identifying a loop, wherein said loop includes a memory load that cannot be disambiguated from a loop upper bound;

locating a memory load instruction within loop body of said loop;

inserting an advance load instruction in preheader of said loop;

replacing said memory load instruction with an advanced load check instruction;

unrolling said loop body; and

generating a cleanup block.

20. The computer readable medium having embodied thereon a computer program in claim **19** wherein said loop is a speculatively counted loop.

21. The computer program being executable by a machine in claim **19** to further perform moving instructions located within a loop from a first location after a memory store instruction to a second location before said memory store instruction in said loop.

22. The computer readable medium having embodied thereon a computer program in claim **19** wherein said cleanup block comprises a rolled copy of original loop body.

23. The computer program being executable by a machine in claim **19** to further perform removing termination tests from between unrolled loop bodies.

24. The computer program being executable by a machine in claim **19** to further perform generating a recovery block for said loop.

25. The computer readable medium having embodied thereon a computer program in claim **19** wherein said cleanup block is a recovery block.

26. A digital processing system having a processor operable to perform:

identifying a loop, said loop having a loop upper bound not proven to be loop invariant;

locating a memory load instruction within loop body of said loop;

inserting an advance load instruction in preheader of said loop;

replacing said memory load instruction with an advanced load check instruction;

unrolling said loop body; and

generating a cleanup block.

27. The digital processing system of claim **26** wherein said loop is a speculatively counted loop.

28. The digital processing system of claim **26** to further perform moving instructions located within said loop from a first location after a memory store instruction to a second location before said memory store instruction in said loop.

29. The digital processing system of claim **26** wherein said cleanup block comprises a rolled copy of original loop body.

30. The digital processing system of claim **26** to further perform removing termination tests from between unrolled loop bodies.

31. The digital processing system of claim **26** to further perform generating a recovery block for said loop.

32. The digital processing system of claim **26** wherein said cleanup block is a recovery block.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 6,539,541 B1
DATED	: March 25,2003
INVENTOR(S)	: Geva

Page 1 of 1

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 2,
Between lines 65 and 66, insert -- $I = I + 3$ --.

Column 10,
Line 5, before "optimize", delete "is".
Line 67, before "factor", delete "to".

Column 15,
Line 27, delete "R3 load y", insert -- *R*3 = *load y* --.

Signed and Sealed this

Tenth Day of June, 2003

[illegible]

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

US005704053A

United States Patent [19]

Santhanam

[11] Patent Number: 5,704,053

[45] Date of Patent: Dec. 30, 1997

[54] EFFICIENT EXPLICIT DATA PREFETCHING ANALYSIS AND CODE GENERATION IN A LOW-LEVEL OPTIMIZER FOR INSERTING PREFETCH INSTRUCTIONS INTO LOOPS OF APPLICATIONS

[75] Inventor: Vatsa Santhanam, Campbell, Calif.

[73] Assignee: Hewlett-Packard Company, Palo Alto, Calif.

[21] Appl. No.: 443,653

[22] Filed: May 18, 1995

[51] Int. Cl.^{6} G06F 9/45

[52] U.S. Cl. 284/383; 395/705

[58] Field of Search 395/425, 375, 395/700, 383, 705

[56] References Cited

U.S. PATENT DOCUMENTS

5,193,167	3/1993	Sites et al.	395/425
5,214,766	5/1993	Liu	395/425
5,377,336	12/1994	Eickemeyer et al.	395/375
5,396,604	3/1995	DeLano et al.	395/375
5,537,620	7/1996	Breternitz, Jr.	395/700

OTHER PUBLICATIONS

Chen, T-F, et al., A Performance Study Of Software & Hardware Data Prefetching Schemes, Apr. 1, 1994, Computer Architecture News, vol. 22, No. 2, pp. 223–232.

Abraham, S G, et al., Predictability Of Load/Store Instruction Latencies, Dec. 1–3, 1993, Proceedings Of The Annual International Symposium On MicroArchitect, Austin, pp. 139–152.

Chi, C-H, et al., Compiler Driven Data Cache Prefetching for High Performance Computers, Proceedings of the regional 10 Annual International Conference, Tenco, Singapore, Aug. 22–26, 1994, vol. 2, No. Conf. 9, pp. 274–278.

Callahan, ., et al., Software Prefetching, ACM Sigplan Notices, vol. 26, No. 4, Apr. 8, 1991, pp. 40–52.

Mowry, T C, et al., Design and Evaluation of a Compiler Algorithm for Prefetching, ACM Sigplan Notices, vol. 27, No. 9, Sep. 1, 1992, pp. 62–73.

Callahan, David, et al., "Software Prefetching", 1991, Association for Computing Machinery.

Klaiber, Alexander C., et al., "An Architecture for Software-Controlled Data Prefetching", May 1991, Int'l Symposium on Computer Architecture.

Chen, William Y., et al., "Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching", Proceedings of the 24th Int'l Symposium on Microarchitecture.

Mowry, Todd C., et al., "Design and Evaluation of a Compiler Algorithm for Prefetching", 1992, Association for Computing Machinery.

Johnson, Eric E., "Working Set Prefetching for Cache Memories".

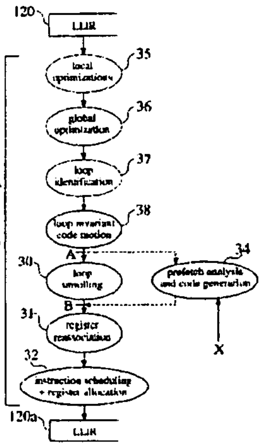
(List continued on next page.)

Primary Examiner—Thomas C. Lee
Assistant Examiner—David Ton

[57] ABSTRACT

A compiler that facilitates efficient insertion of explicit data prefetch instructions into loop structures within applications uses simple address expression analysis to determine data prefetching requirements. Analysis and explicit data cache prefetch instruction insertion are performed by the compiler in a machine-instruction level optimizer to provide access to more accurate expected loop iteration latency information. Such prefetch instruction insertion strategy tolerates worst-case alignment of user data structures relative to data cache lines. Execution profiles from previous runs of an application are exploited in the insertion of prefetch instructions into loops with internal control flow. Cache line reuse patterns across loop iterations are recognized to eliminate unnecessary prefetch instructions. The prefetch insertion algorithm is integrated with other low-level optimization phases, such as loop unrolling, register reassociation, and instruction scheduling. An alternative embodiment of the compiler limits the insertion of explicit prefetch instructions to those situations where the lower bound on the achievable loop iteration latency is unlikely to be increased as a result of the insertion.

17 Claims, 9 Drawing Sheets



OTHER PUBLICATIONS

Gornish, Edward H., et al., "Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies", 1990, Association for Computing Machinery.

Gupta, Anoop, et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques", 1991, Association for Computing Machinery.

Fu, John W. C., et al., "Data Prefetching in Multiprocessor Vector Cache Memories", 1991, Association for Computing Machinery.

Chen, Tien-Fu, et al., "Reducing Memory Latency via Non-blocking and Prefetching Cahces", 1992, Association for Computing Machinery.

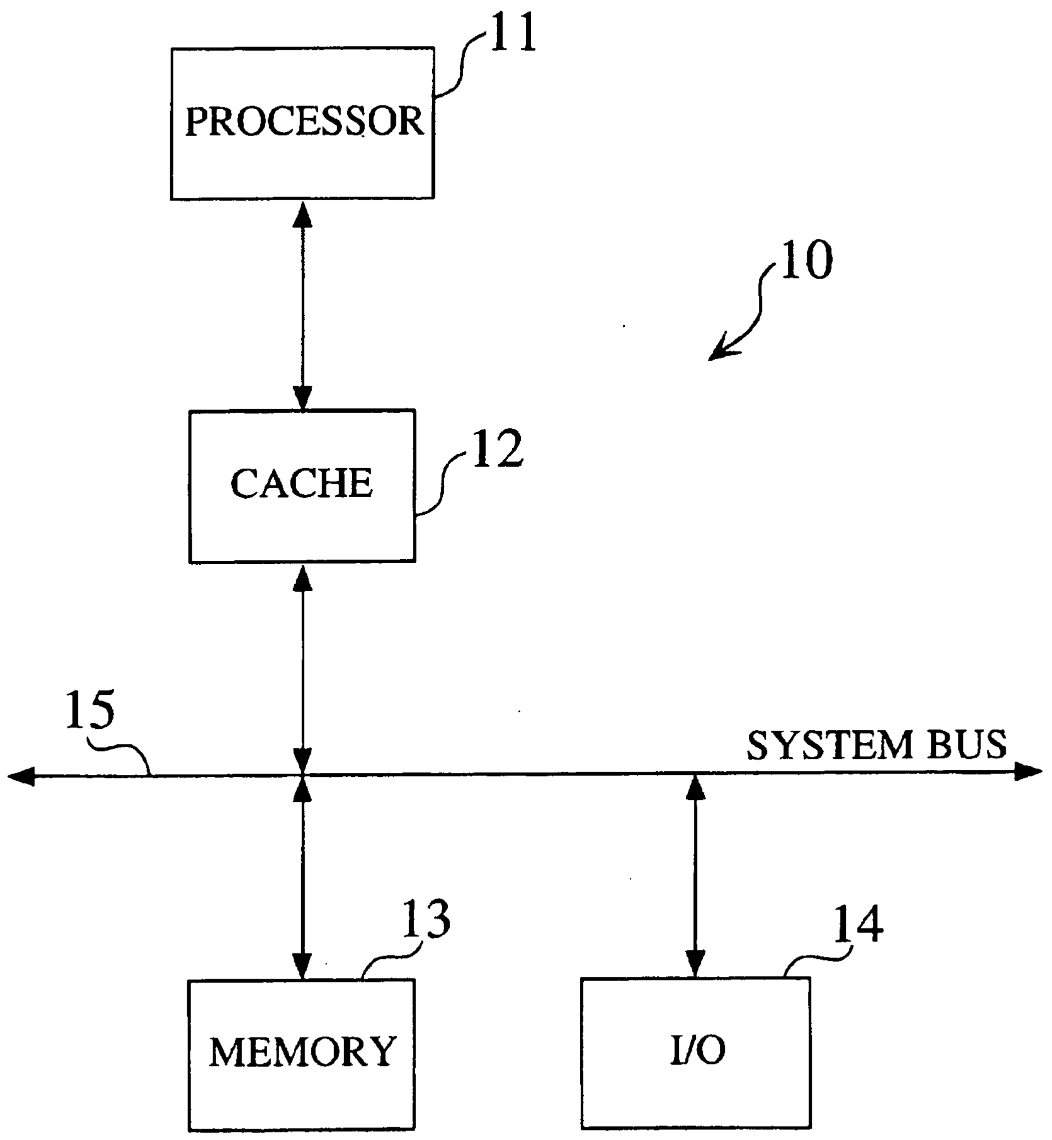


Fig. 1 (PRIOR ART)

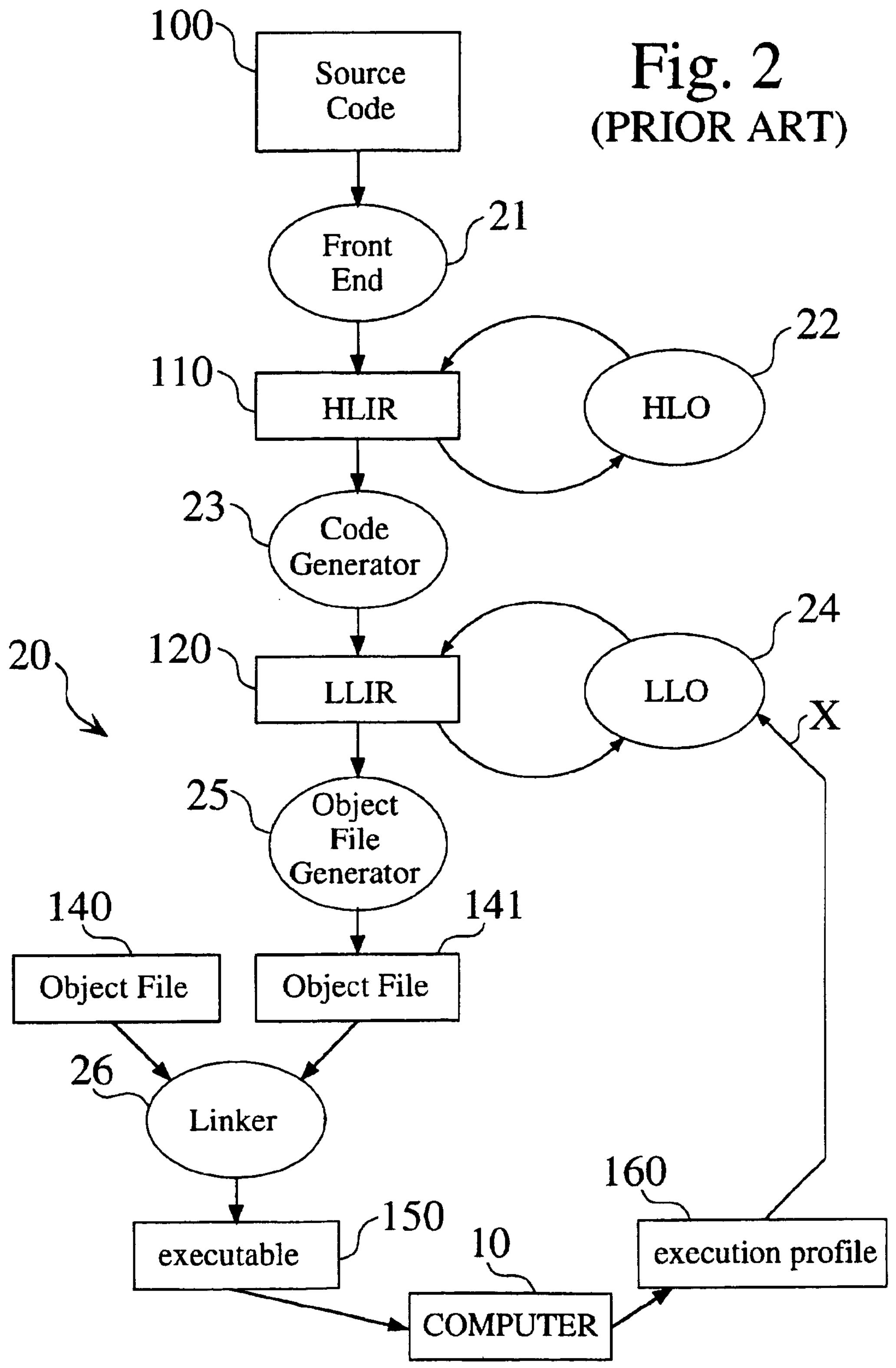


Fig. 2
(PRIOR ART)

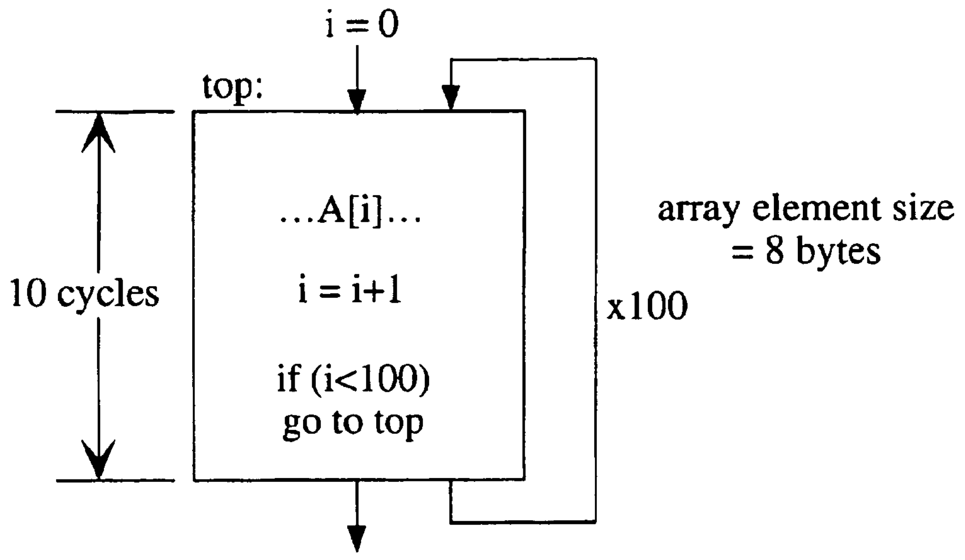


Fig. 3
(PRIOR ART)

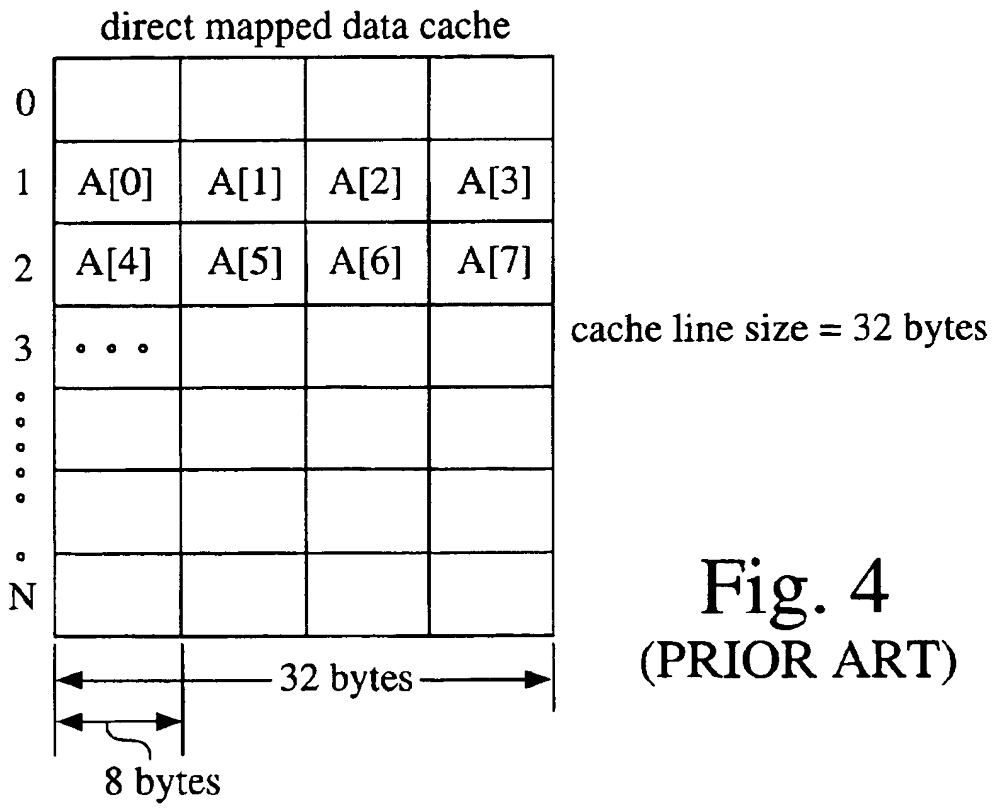


Fig. 4
(PRIOR ART)

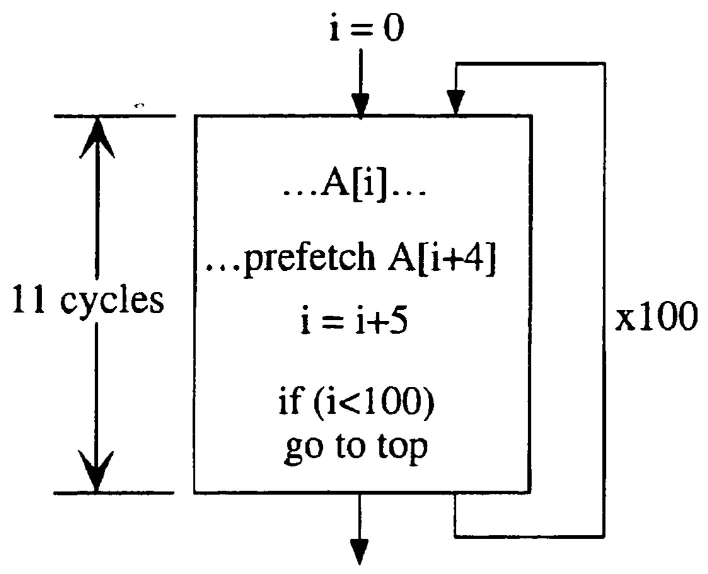


Fig. 5 (PRIOR ART)

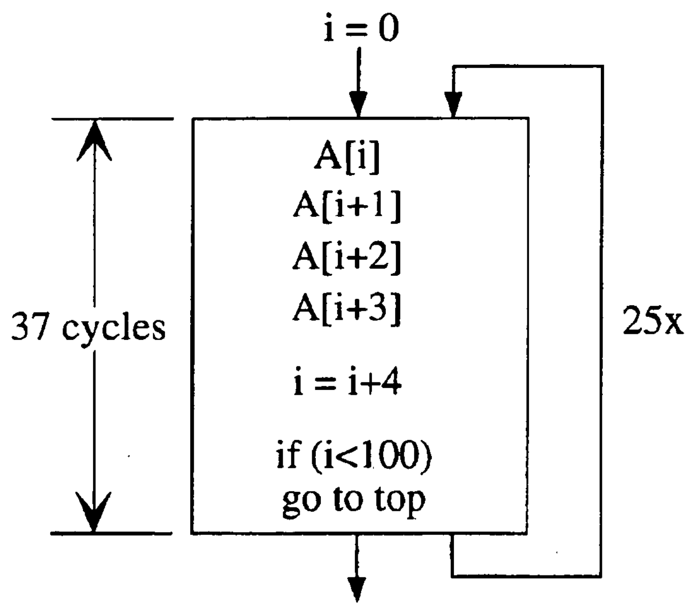


Fig. 6 (PRIOR ART)

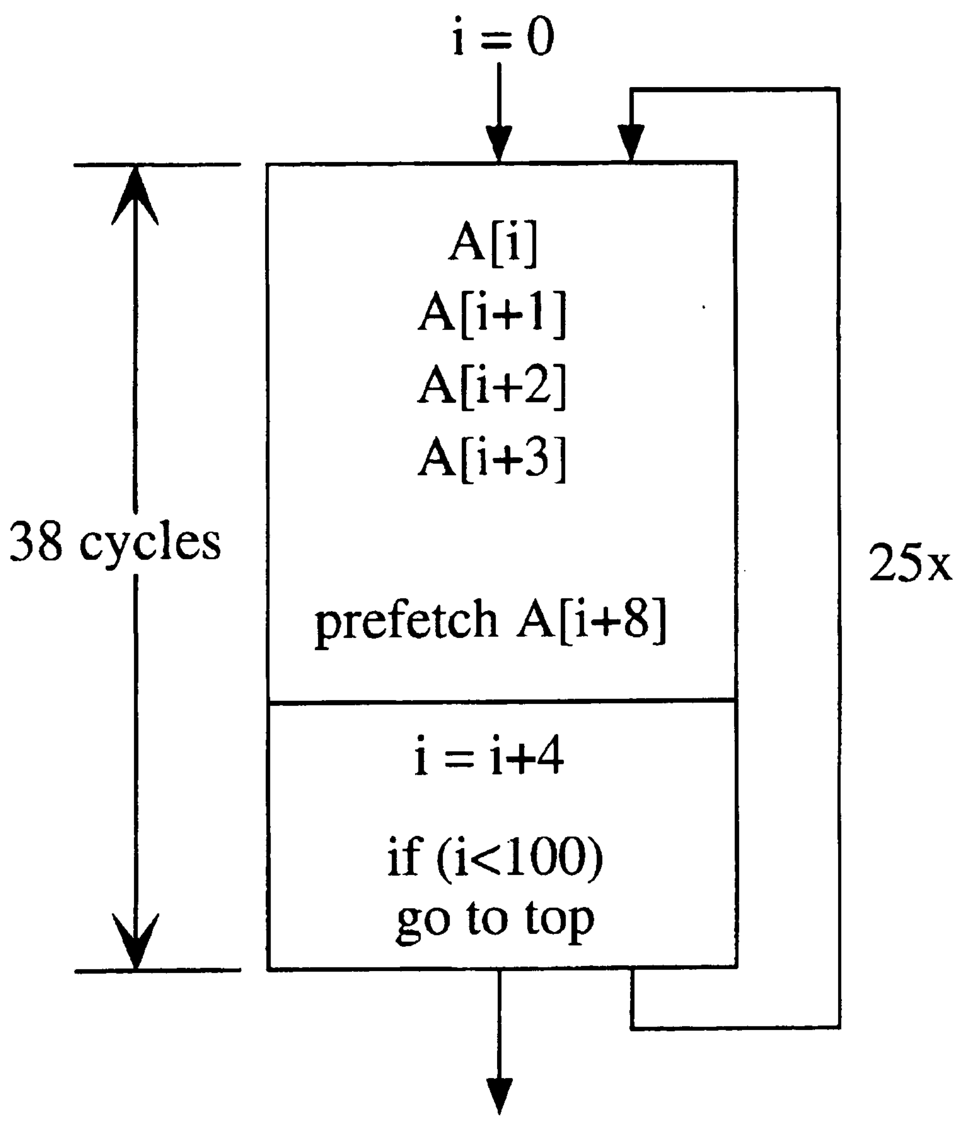


Fig. 7
(PRIOR ART)

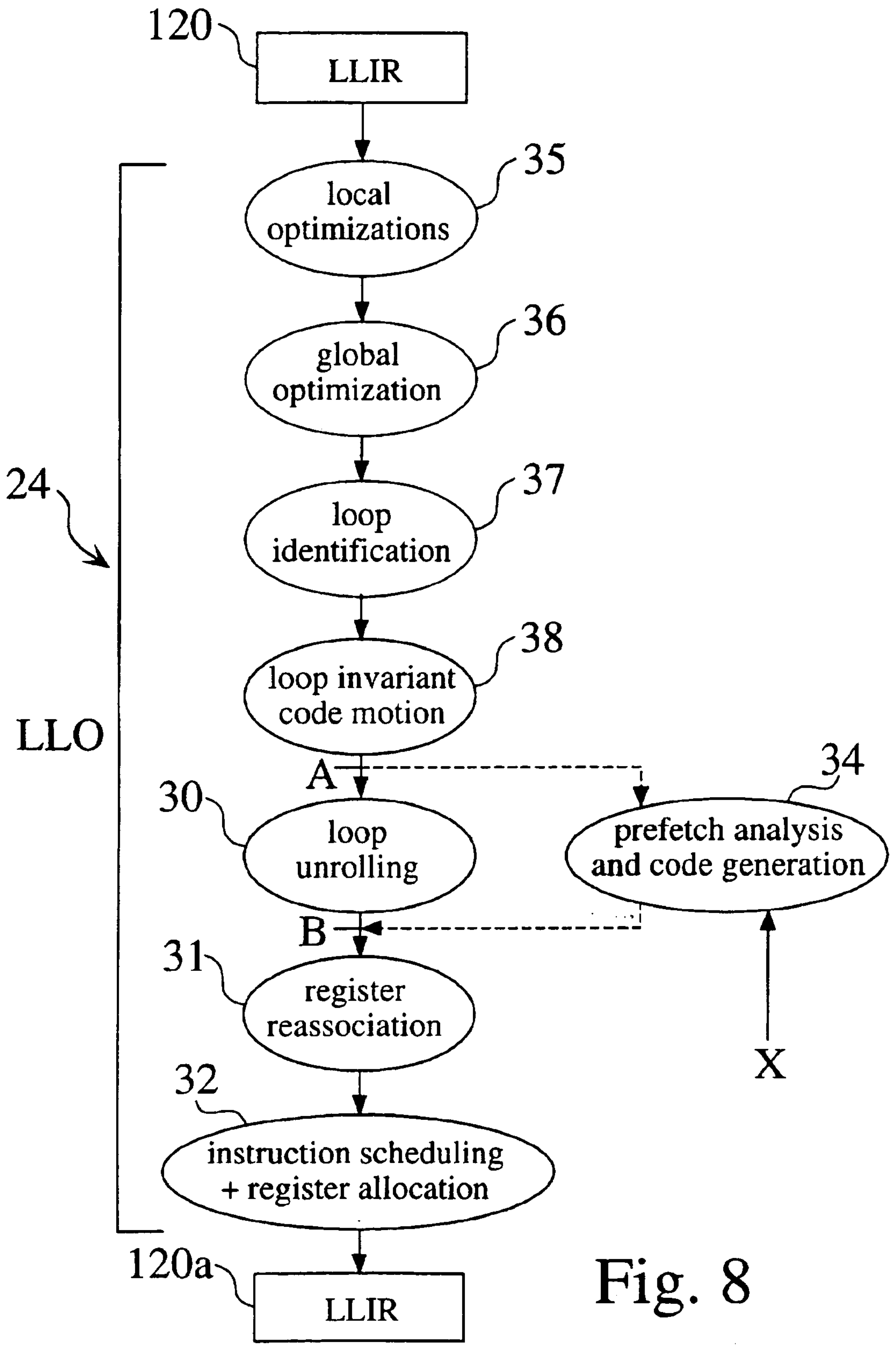

120
LLIR
35
local optimizations
36
global optimization
24
37
loop identification
38
loop invariant code motion
LLO
A
30
loop unrolling
34
prefetch analysis and code generation
B
31
register reassociation
X
32
instruction scheduling + register allocation
120a
LLIR

Fig. 8

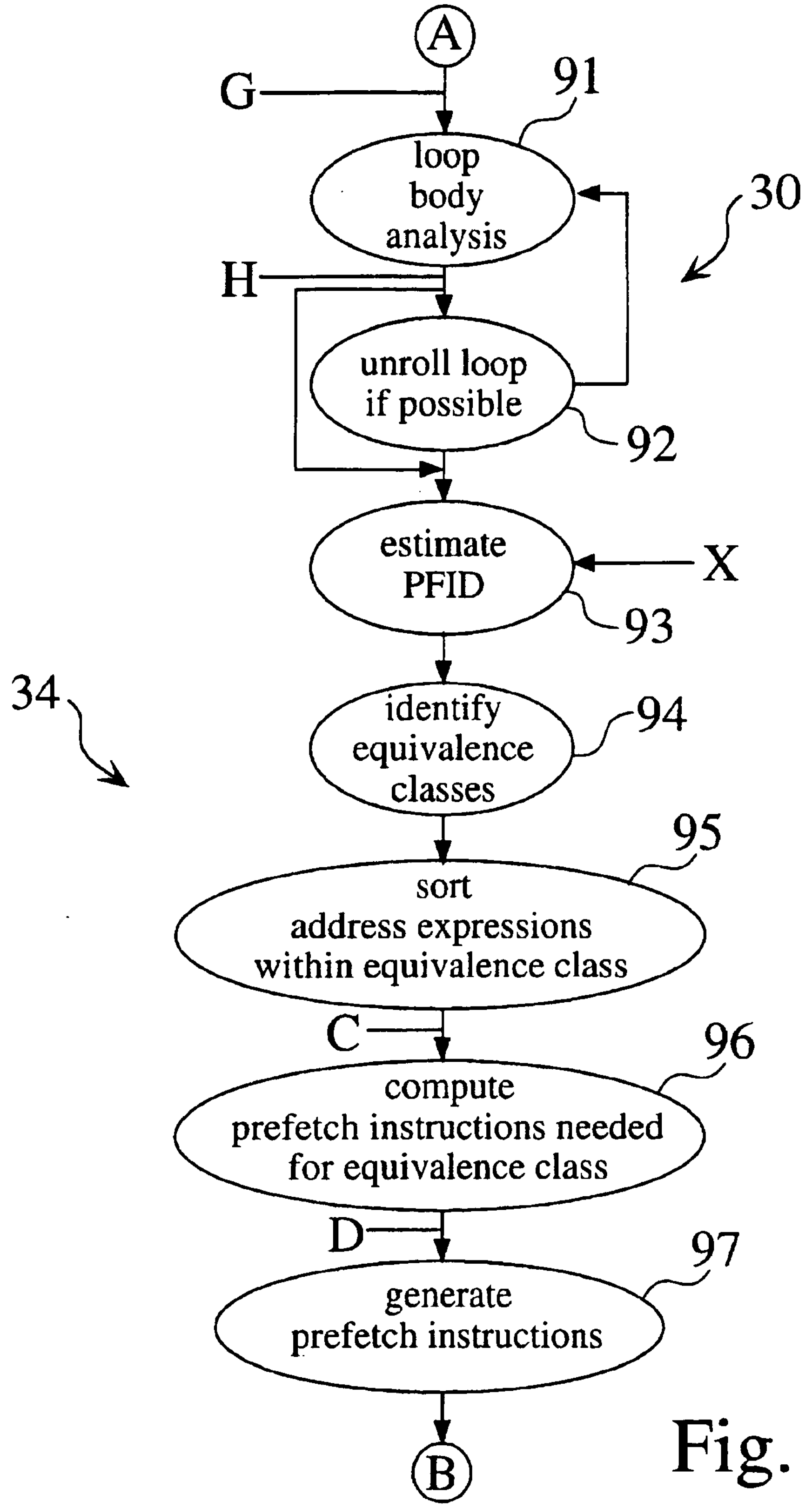

A
G
91
loop body analysis
30
H
unroll loop if possible
92
estimate PFID
X
93
34
identify equivalence classes
94
95
sort address expressions within equivalence class
C
96
compute prefetch instructions needed for equivalence class
D
97
generate prefetch instructions
B

Fig. 9

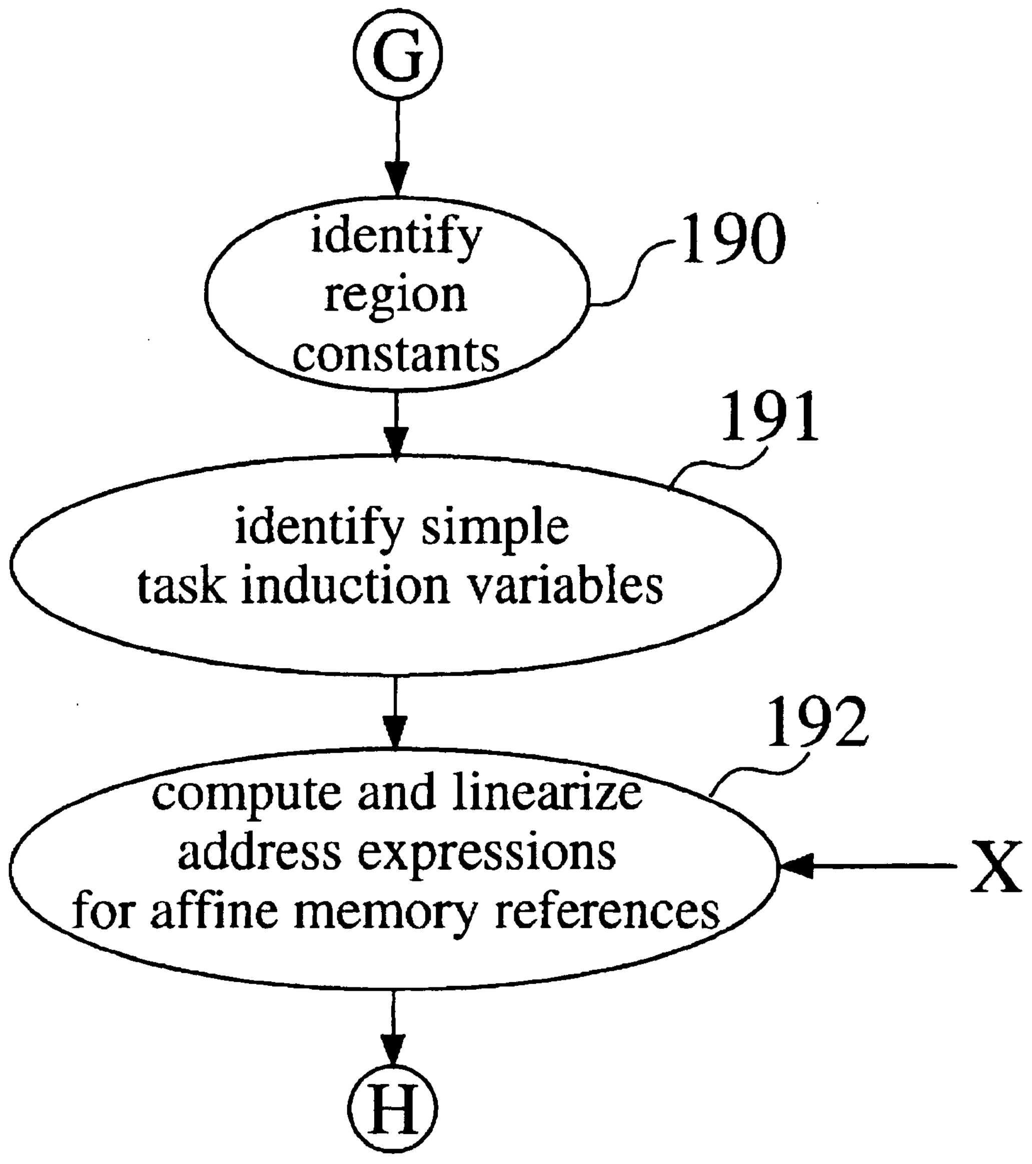

G
identify region constants
190
191
identify simple task induction variables
192
compute and linearize address expressions for affine memory references
X
H

Fig. 10

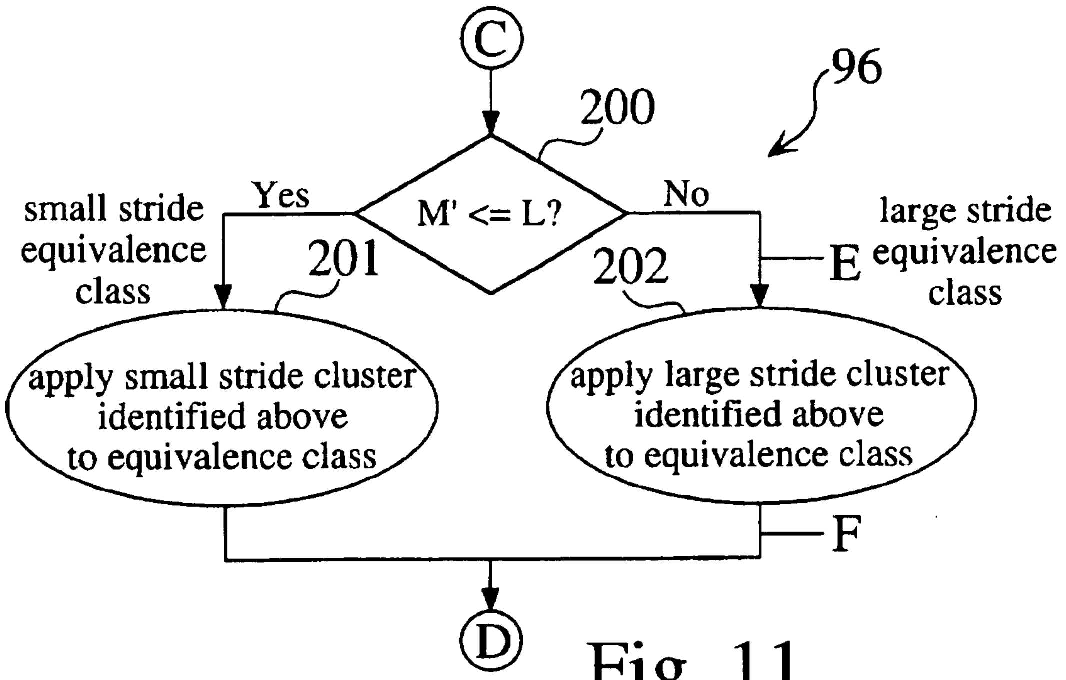

C
96
200
M' <= L?
Yes
No
small stride equivalence class
large stride equivalence class
201
202
E
apply small stride cluster identified above to equivalence class
apply large stride cluster identified above to equivalence class
F
D

Fig. 11

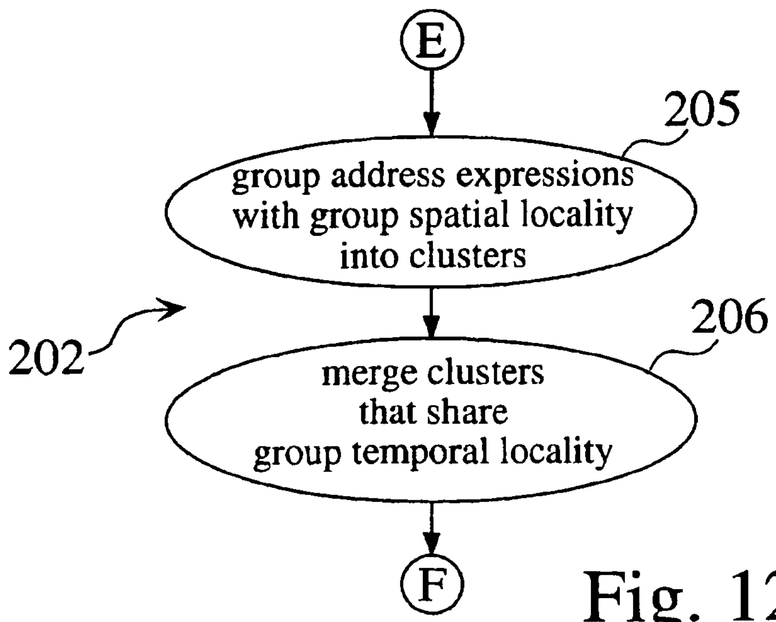

E
205
group address expressions with group spatial locality into clusters
202
206
merge clusters that share group temporal locality
F

Fig. 12

EFFICIENT EXPLICIT DATA PREFETCHING ANALYSIS AND CODE GENERATION IN A LOW-LEVEL OPTIMIZER FOR INSERTING PREFETCH INSTRUCTIONS INTO LOOPS OF APPLICATIONS

BACKGROUND OF THE INVENTION

1. Technical Field

The invention relates to techniques for reducing data cache overhead in a computer system. More particularly, the invention relates to compiler-related techniques that are useful for reducing data cache overhead.

2. Description of the Prior Art

Data cache misses (described in greater detail below) can account for a significant portion of an application program's execution time on modern processors. This is particularly true in the case of scientific applications that manipulate large data structures which run on high frequency processors having long memory latencies. With increasing mismatch between processor and memory, the high penalty of cache misses has become and continues to be a dominant performance limiter of microprocessors. Increasing the cache size is one way to reduce cache misses. However, because the size of many numerical applications is also growing rapidly from generation to generation, the first level cache may not always be large enough to capture critical working sets.

Most modern computer systems employ such caches to bridge the gap between memory and processor speeds. However, despite high cache hit ratios, the cost of cache misses in high frequency processors can significantly degrade run-time performance. To illustrate this point, a plausible scenario has been suggested where a cache miss penalty is 100 processor cycles and a data reference occurs every four cycles. See, for example Alexander C. Klaiber, Henry M. Levy, An Architecture for Software-Controlled Data Prefetching, Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991. Even assuming a cache hit ratio of 99%, the processor is stalled for memory 20% of the time.

One way to ameliorate the high overhead of data cache misses is to overlap the fetching of data from memory to the data cache with other useful computations. Certain high-performance superscalar microprocessors are able to achieve some degree of overlap between data cache miss handling and processor computation automatically through out-of-order instruction execution, facilitated by instruction queues capable of holding renamed register results, in conjunction with a split-transaction memory bus (e.g. such microprocessors as the Silicon Graphics T5, Hewlett-Packard PA8000, and Sun Ultrasparc). However, the degree of overlap typically achieved is insufficient to fully cover an external data cache miss latency.

Some of these microprocessors support explicit data prefetch instructions that may be used to reduce the high overhead of data cache misses more effectively. Such instructions are typically defined to initiate data cache miss handling without holding up instruction execution until the referenced data is retrieved from memory.

By inserting explicit data cache prefetch instructions into the code stream, a compiler can help ameliorate the high cost of data cache misses. However, this approach must be implemented judiciously because explicit cache prefetch instructions, in general, increase the dynamic path length of an application, and the added overhead may not be offset by a corresponding decrease in data cache miss overhead.

There is much published literature on cache design trade-offs, and hardware approaches to improving cache performance. Comparatively, however, there is much less literature on improving cache performance through software-controlled active cache management. The few papers that discuss software-controlled data prefetching to improve cache performance include Todd C. Mowry, Monica S. Lam, Anoop Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992; Alexander C. Klaiber, Henry M. Levy, An Architecture for Software-Controlled Data Prefetching, Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991 (an approach based on hand analysis); and Software Prefetching, David Callahan, Ken Kennedy, Allan Porterfield, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991 (where a prefetch instruction is added for each loop body memory reference without considering or exploiting cache line re-use, such that there is no selectivity; and where the prefetch insertion is performed at the source-code level, such that there is little integration with other compiler optimization phases; additionally, because the analysis is done at the source code level, it is difficult to estimate the prefetch iteration distance (PFID), i.e. the PFID used is always one loop iteration, which may be insufficient to hide the full cache miss latency).

These papers concentrate on explicit prefetches for subscripted variables that are referenced in loops. They do not discuss insertion of explicit prefetch instructions into straight-line code for scalar or indirect memory references. Furthermore, it is generally assumed that the arrays of interest are all aligned on cache line boundaries.

There are some general observations that are more or less common to the different studies of software-controlled data prefetching. One such observation is that data prefetching does not come for free. Specifically, explicit prefetches use up instruction issue bandwidth. In addition to the prefetch instruction itself, typically one or more instructions are needed to compute the address of the memory location to be prefetched. Recycling the computed prefetch address for the actual reference can involve tying up registers for extended lifetimes. The increased register pressure can result in the introduction of spill code in expensive loops. This can offset the expected performance gains due to prefetching.

A simple prefetch strategy, such as the one proposed by David Callahan, Ken Kennedy, Allan Porterfield, Software Prefetching, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, can wastefully increase the number of executed instructions through multiple prefetch requests for lines already in the data cache.

Another important consideration cited by the different papers on software-controlled data prefetching is the actual placement of the prefetch instructions. If a prefetch is issued too close time-wise to the memory reference that needs to access the prefetched data, the prefetched data may not be available in time to avoid a CPU stall. On the other hand, if the prefetch is issued too early, there is a possibility of the prefetched line being displaced from the cache prematurely.

Todd C. Mowry, Monica S. Lam, Anoop Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating

Systems, October 1992, discuss the notion of identifying a prefetch predicate and the leading reference amongst multiple references to an array to facilitate selective prefetching. This paper also discusses the interaction of data prefetching with other compiler transformations, specifically cache blocking and software pipelining. The prefetching algorithm disclosed is effective at reducing explicit data prefetch overhead. One shortcoming with this approach is that it relies on reuse and locality analysis that is rather complex. The analysis is done in the context of a high-level optimizer, which makes it difficult to estimate the prefetch iteration distance because the effects of downstream compiler components (e.g. code generator and low-level optimizer) on the loop body are unknown. It is also unclear how cache line alignment of prefetched data structures is accounted for when memory strides are greater than the cache line size. Also, it is unclear whether unnecessary prefetches are inserted for certain types of data reuse patterns. For instance, for the following C code fragment, the disclosed algorithm may actually insert three prefetches when two would be sufficient to ensure full miss coverage.

```
int A[100][100];
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
    {
        . . . A[j-1][i] . . .
        . . . A[j][i+1] . . .
        . . . A[j+1][i-1] . . .
    }
```

SUMMARY OF THE INVENTION

Assume that a target processor supports a data cache line prefetch instruction with the following characteristics:

- It allows a memory address to be specified much like in an ordinary load or store instruction;
- If the memory referenced by the prefetch instruction is not found in the data cache, the processor causes the referenced memory location to be retrieved from lower levels of the memory hierarchy without stalling the execution of other instructions in the processor's execution pipelines; and
- The processor does not signal an exception even when the memory address specified by a prefetch instruction is invalid.

The current invention provides a new compiler for such a processor that facilitates efficient insertion of explicit data prefetch instructions into loops within application programs. The compiler uses simple subscript expression analysis to determine data prefetching requirements. Analysis and explicit data cache prefetch instruction insertion are performed by the compiler in a machine instruction level optimizer to provide access to more accurate expected loop iteration latency information.

Such a prefetch instruction insertion strategy tolerates worst case alignment of user data structures relative to data cache lines. Execution profiles from previous runs of an application are exploited in the insertion of prefetch instructions into loops with internal control flow. Cache line reuse patterns across loop iterations are recognized to eliminate unnecessary prefetch instructions. The prefetch insertion algorithm is integrated with other low level optimization phases, such as loop unrolling, register reassociation, and instruction scheduling.

An alternative embodiment of the compiler limits the insertion of explicit prefetch instructions to those situations

where the lower bound on the achievable loop iteration latency is unlikely to be increased as a result of the insertion.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block schematic diagram of a uniprocessor computer architecture including a processor cache;

FIG. 2 is a block schematic diagram of a modern software compiler;

FIG. 3 is a schematic representation of a loop;

FIG. 4 is a schematic representation of a direct mapped data cache;

FIG. 5 is a schematic representation of a loop, including a prefetch instruction;

FIG. 6 is a schematic representation of a loop that has been unrolled four times;

FIG. 7 is a schematic representation of an unrolled loop, including a prefetch instruction;

FIG. 8 is a block diagram showing a low level optimizer for a compiler, including a prefetch driver according to the invention;

FIG. 9 is a block diagram of a prefetch driver according to the invention;

FIG. 10 is a block diagram of a loop body analysis module according to the invention;

FIG. 11 is a block diagram of a module that is used to compute prefetch instruction needed for equivalence class according to the invention; and

FIG. 12 is a block diagram of a module that applies a large stride cluster identifier to an equivalence class according to the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention provides a new compiler that facilitates efficient insertion of explicit data prefetch instructions into loops within applications. FIG. 1 is a block schematic diagram of a uniprocessor computer architecture 10 including a processor cache. In the figure, a processor 11 includes a cache 12 which is in communication with a system bus 15. A system memory 13 and one or more I/O devices 14 are also in communication with the system bus.

FIG. 2 is a block schematic diagram of a software compiler 20, for example as may be used in connection with the computer architecture 10 shown in FIG. 1. The compiler Front End component 21 reads a source code file (100) and translates it into a high level intermediate representation (110). A high level optimizer 22 optimizes the high level intermediate representation 110 into a more efficient form. A code generator 23 translates the optimized high level intermediate representation to a low level intermediate representation (120). The low level optimizer 24 converts the low level intermediate representation (120) into a more efficient (machine-executable) form. Finally, an object file generator 25 writes out the optimized low-level intermediate representation into an object files (141). The object file (141) is processed along with other object files (140) by a linker 26 to produce an executable file (150), which can be run on the computer 10. In the invention described herein, it is assumed that the executable file (150) can be instrumented by the compiler (20) and linker (26) so that when it is run on the computer 10, an an execution profile (160) may be generated, which can then be used by the low level optimizer 24 to better optimize the low-level intermediate representation (120). The compiler 20 is discussed in greater detail below.

in contrast to previous approaches to the cache miss problem discussed above (see Todd C. Mowry, Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis, Dept. of Electrical Engineering, Stanford University, March 1994; D. Callahan, K. Kennedy, A. Porterfield, Software Prefetching, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 40–52, April 1991; and W. Y. Chen, S. A. Mahlke, P. P. Chang, W. W. Hwu, Data access microarchitectures for superscalar processors with compiler-assisted data prefetching, Proceedings of Microcomputing 24, 1991) the new compiler has the following unique attributes:

Simple subscript expression analysis is used to determine data prefetching requirements, as opposed to sophisticated reuse/dependence analysis.

Subscript expression analysis and explicit data cache prefetch instruction insertion are performed by the compiler in a low-level, i.e. machine instruction level, optimizer. A principal advantage of this approach is access to more accurate expected loop iteration latency information.

The prefetch instruction insertion strategy tolerates worst case alignment of user data structures relative to data cache lines.

Execution profiles from previous runs of an application are exploited in the insertion of prefetch instructions into loops with internal control flow.

Cache line reuse patterns across loop iterations are recognized to eliminate unnecessary prefetch instructions.

The prefetch insertion algorithm is integrated with other low level optimization phases, such as loop unrolling, register reassociation, and instruction scheduling.

An alternative embodiment of the new compiler also limits the insertion of explicit prefetch instructions to those situations where the lower bound on the achievable loop iteration latency is unlikely to be increased as a result of the insertion.

The new compiler yields significant performance improvements for some industry-standard performance benchmarks on simulations of the Hewlett-Packard Company (Palo Alto, Calif.) PA-8000 processor.

The following discussion explains compiler operation in the context of a loop within an application program. Loops are readily recognized as a sequence of code that is iteratively executed some number of times. The sequence of such operations is predictable because the same set of operations is repeated for each iteration of the loop. It is common practice in an application program to maintain an index variable for each loop that is provided with an initial value, and that is incremented by a constant amount for each loop iteration until the index variable reaches a final value. The index variable is often used to address elements of arrays that correspond to a regular sequence of memory locations. Such array references by a loop constitute a significant portion of cache misses in scientific applications.

In the compiler, it has been found that the low level optimizer component of a compiler is in a good position to deduce the number of cycles required by a stretch of code that is repetitively executed. As discussed above, the concept of prefetching is not new.

Nonetheless it is helpful to explain prefetching at this point. For example, assume that the time that it takes to get a data item back from main memory to cache is 100 cycles, during which time, the processor must wait idly before it can operate on the data. To avoid wasting idle processor cycles

on account of data cache misses, it it is desirable to initiate retrieval of data that is not likely to be found in the cache, in advance of such data being needed by the processor. The compiler can predict which data is needed in advance for loops that access array elements in a regular fashion. The compiler can then insert prefetch instructions into loops such that array elements that are likely to be needed in future loop iterations are retrieved from memory ahead of time. Ideally, the number of iterations in advance that array elements are prefetched is such that by the time the array element is actually required by the processor, the array element is retrieved from memory and placed in the data cache (if it was not there to begin with).

In prior art approaches to prefetching, cache alignment is a problem. Another known problem is the overhead of the prefetch instruction itself. These are very important problems. Run time array dimensioning is yet another problem that must be addressed.

For example, in FIG. 3 a loop is shown that has a loop execution time of 10 cycles and that iterates 100 times, accessing an 8-byte array element on each iteration. If there are no cache misses, the total loop execution time is 1000 cycles. In FIG. 4, a direct mapped data cache is shown where the cache line size is 32 bytes, each line capable of holding 4 contiguous 8-byte array elements. For the loop of FIG. 3, it is assumed that a cache miss occurs every fourth iteration (on every cache line crossing), which means that 25 data cache misses will occur for the whole loop. If it takes 40 cycles to service each cache miss, the total loop execution time becomes 2000 cycles, i.e. 1000 cycles for just executing the loop instructions +25×40 cycles, or another 1000 cycles, for the cache misses.

If known prefetch techniques are used, then for the example of FIGS. 3 and 4 cache misses can be covered if a prefetch distance of four is chosen. In FIG. 5, a prefetch instruction is shown inserted into the loop of FIG. 3. As can be seen, the use of a prefetch instruction can eliminate most cache misses, thereby saving significant execution time. However, a prefetch instruction requires execution time. In the example herein, each iteration of the loop requires a prefetch instruction, which can be assumed to take an extra cycle. Therefore, for a loop that iterates 100 times, 100 cycles must be added to the execution time to account for prefetching.

Additionally, the first iteration of the loop incurs a cache miss, which in the example herein requires 40 cycles. Accordingly, prefetching avoids most cache misses, such that execution time is reduced to 1140 cycles, i.e. 1000 cycles to execute the original loop instructions +100 cycles for the prefetch instructions+40 cycles for the initial cache miss before the first prefetch instruction is executed. Thereafter, the prefetch instructions overlap the 40-cycle data cache miss service time with the execution of four (11-cycle) loop iterations.

Unfortunately, each time through the loop a new prefetch instruction is executed. Where the unit of transfer between the main memory and the cache is a cache line, some of the prefetches are redundant because a prefetch for a particular array location may refer to the same cache line as the prefetch for subsequent array locations. This redundancy occurs because there are adjacent array locations in the same cache line, and the system is issuing a redundant instruction to the memory system to retrieve the same cache line multiple times. Typically, computer systems that support this type of prefetch instruction track the instructions to determine if a requested address to prefetch a cache line matches a later prefetch to the same cache line. In such event, the second prefetch request to main memory is dropped.

However, even though redundant prefetches typically get dropped, it is nonetheless important that prefetch instructions that refer to the same cache line are not executed multiple times because the prefetch instruction itself takes up some compute time. The processor must fetch and execute the prefetch instruction, understand what data address the instruction refers to, and then access the data cache to check if the data are already cache-resident.

Note that the compiler is responsible for inserting prefetch instructions into a loop body that specify the memory address of data items that will be accessed in the future. The memory address is determined based on the number of loop iterations in advance (i.e. the prefetch iteration distance or PFID) that data items need to be prefetched to fully hide the time required to service potential data cache misses. The PFID is determined taking into account the nature of the loop body instructions and characteristics of the target processor and memory system. For instance, for a "short" loop, e.g. one that takes only two cycles per iteration to execute, the PFID would need to be 50 in order to accomodate a 100-cycle data cache miss latency.

The key to efficient data prefetching then is to overlap the computer's execution of the instructions in a piece of code, such as a loop, with the time it takes to retrieve the data from the memory and place it to the processor cache, and do this in a way that avoids redundant prefetches.

Ideally, cache miss overhead is completely eliminated by inserting prefetch instructions judiciously. Referring back to the example above where the loop executes 100 iterations, with each iteration taking 11 cycles each (10 cycles for the original loop body instructions +1 cycle for the prefetch instruction +40 cycles for an initial cache miss before prefetching starts), the time it takes to run the loop is only 1140 cycles, which is much better than the 2000 cycles of the example above in FIG. 4.

However, 1140 cycles is still not quite as good as 1000 cycles. One way to increase further the savings in processor execution time is by using a well known technique referred to loop unrolling. In loop unrolling, the body of the loop is replicated. This reduces the number of times the loop is executed by a factor that is equal to the number of replications, although each time the code is executed there are more instruction to exectue. Thus, in loop unrolling exactly the same amount of work is accomplished, but the loop is now reorganized.

Note however, that because the loop closing branch doesn't not have to be replicated as many times as the loop body is unrolled (in fact an unrolled loop typically needs just one loop-closing branch), loop unrolling can by itself result in improved performance.

For example, FIG. 6 shows the loop of FIG. 3 after the loop has been unrolled four times. Thus, instead of executing the loop 100 times, the loop is executed 25 times. Assume that a loop-closing branch takes 1-cycle to execute. Each iteration in the unrolled loop would then require 37 cycles (4×9 cycles +1 cycle) and the total loop execution time is equal to (25 iterations×37 cycles)+(25 iterations×40 cycles/cache miss)=1925 cycles.

In the context of the invention and per the example above, if each iteration of the unrolled loop requires 37 cycles, where the loop is unrolled four times, it is necessary to prefetch data two iterations ahead (since 1 iteration ahead is insufficient to accomodate a 40 cycle cache miss latency). If the prefetch instruction is put at the bottom of the loop, then the loop is executed before a prefetch is performed. This does not provide optimum operation of the loop. Thus, the placement of the prefetch instruction is critical. It is therefore necessary to place the prefetch instruction at a point that provides sufficient time for a prefetch before the loop completes execution. For example, if the prefetch is placed at the top of the loop, then the loop does the same amount of work, but more effectively overlaps the time to service a possible data cache miss for subsequent iterations with the computation performed in the current iteration.

For the example above, where there are 100 iterations of a 10-cycle loop that takes a total of 1000 cycles, the prefetch instructions cost 100 cycles +a 40 cycle cache miss for the first iteration. As a result, the execution of the loop is reduced from 2000 cycles to 1140 cycles. By adding loop unrolling, in this example where the loop is unrolled by a factor of four (see FIGS. 6 and 7), each iteration of the loop may take 38 cycles (37 cycles+1 cycle for the prefetch instruction). Thus, execution time for the loop is equal to 38 cycles×25 iterations+80 cycles for two cache misses before prefetching begins =1030 cycles. Thus, it is clear that the techniques disclosed herein produce a substantial improvement in the execution time for a loop.

Note that in some cases the prefetch instruction may not cost any additional cycles to execute. This is because many modern processors are superscalar, i.e. they can execute multiple instructions in one cycle, e.g. a load with an add. Prefetch instructions are similar to a load because they refer to memory. Thus, if there are several adds in a loop, adding one extra prefetch does not increase the time necessary to execute each iteration of the loop because the prefetch instruction is executed in parallel with the add instruction.

One important feature of the invention identifies loops and access patterns to allow a determination of how many cycles are devoted to loop iterations, and therefore allows insertion of the prefetch instruction to a location of an array that is sufficiently far in advance to make sure that the miss time is minimized. One problem is that loops can be coded in many different ways. It is therefore necessary to recognize different types of loops. For example, there are some loops that are not always handled by prefetching.

In the invention, the compiler translates the higher level application into an instruction stream that the processor executes, where the compiler inserts prefetches at opportune points into the instruction stream that effects data retrieval from main memory into the cache in advance of when that data item is actually needed. The compiler anticipates that a particular data item is going to be needed at a particular time, rather than letting the processor execute blindly until it gets to that point in time, where the processor stalls and waits for several dozens of cycles until that item is fetched from main memory.

One advantage in letting the processor continue executing the other instructions which may have nothing to do with memory is that the system can achieve an overlap. While the access time between processor and cache is typically 1 to 5 cycles, the retrieval time from cache to memory is often on the order of 10 to 100 cycles. When the processor actually gets to the point where the data item is needed, it is not necessary to wait for a cache miss that takes 100 processor cycles. Thus, instead of waiting 100 processor cycles, it may only be necessary to wait for 20 cycles because 80 cycles worth of look up time is hidden or overlapped with the previous execution.

The invention is preferably implemented in the low level optimizer of the compiler to insert prefetch instructions at opportune points in the code. In particular, the invention inserts prefetch instructions into loops. One advantage of inserting prefetch instructions into loops is that the data reference pattern of a loop tends to be regular and the compiler is better able to predict the kind of memory items that are likely to be required in the future, where the future is not the current iteration of the loop, but five or six iterations in the future. As discussed above, this depends on the characteristics of the instructions found within the loop. It is therefore necessary to vary the point in time that the prefetch request is actually issued, based on the expected latency of a loop iteration.

The compiler is the piece of software that translates source code, such as C, BASIC, or FORTRAN, into a binary image that actually runs on a machine. Typically the compiler consists of multiple distinct phases, as discussed above in connection with FIG. 2. One phase is referred to as the front end, and is responsible for checking the syntactic correctness of the source code. If the compiler is a C compiler, it is necessary to make sure that the code is legal C code. There is also a code generation phase, and the interface between the front-end and the code generator is a high level intermediate representation. The high level intermediate representation is a more refined series of instructions that need to be carried out. For instance, a loop might be coded at the source level as:

```
for(I=0, I<10, I=I+1),
```

which might in fact be broken down into a series of steps, e.g. each time through the loop first load up I and check it against 10 to decide whether to execute the next iteration. A code generator takes this high level intermediate representation and transforms it into a low level intermediate representation. This is much closer to the actual instructions that the computer understands. In terms of improving the quality of the intermediate representations, a low level intermediate representation generated by a code generator is typically fed into a low level optimizer.

An optimizer component of a compiler must preserve the program semantics (i.e. the meaning of the instructions that are translated from source code to an high level intermediate representation, and thence to a low level intermediate representation and ultimately an executable file,) but may rewrite or transform the code in a way that allows the computer to execute an "equivalent" set of instructions to be executed in less time.

Modern compilers are structured with a high level optimizer (HLO) that typically operates on a high level intermediate representation and substitutes in its place a more efficient high level intermediate representation of a particular program that is typically shorter. For example, an HLO might eliminate redundant computations.

With the low level optimizer (LLO), the over-arching objectives are largely the same as the HLO, except that the LLO operates on a representation of the program that is much closer to what the machine actually understands. Uniquely, the invention performs prefetch analysis and prefetch instruction generation in the context of a low level optimizer. At this level there are not any semantic annotations, but merely instructions, such as add, load, and store. The compiler herein identifies repetitive code segments, such as loops, for prefetch instruction generation in the context of a low level optimizer.

The analysis that the compiler herein uses is simpler than that of the prior art. Additionally, because the invention operates in the context of a low level optimizer on raw instructions, it is much easier to estimate how many processor cycles a loop iteration requires, and therefore how many iterations in advance a data prefetch instruction should be inserted into the code.

There are many different organizations that are possible for a data cache, but one possible organization that is not all that uncommon is to organize it in terms of a series of direct-mapped cache lines. Each cache line might be able to hold up to 32 bytes of data, such that the unit of transfer between main memory and the cache is in chunks of 32 bytes. The processor can make a request to the memory system, and thence data is placed into a well defined location in the cache to allow the processor to retrieve the data from that location. If the cache in this example is 32,000 bytes in size, then it has 1000 lines, each line holding 32 bytes. Any explicit data prefetching compiler ultimately has to insert the hardware prefetch instructions into the low level code representation. One distinguishing feature of the current invention is that the analysis required to insert prefetch instructions efficiently is also done in the context of a low level optimizer. Moreover, the prefetch instruction insertion is done a manner that is synergistic with other low level optimization, such as loop unrolling, register reassociation, and instruction scheduling.

The invention herein resides within the domain of the low level optimizer 24. FIG. 8 is a block diagram showing a low level optimizer for a compiler, including a prefetch driver 34 according to the invention.

The low level optimizer 24 in accordance with the preferred embodiment of the invention may include any combination of known optimization techniques, such as those that provide for local optimization 35, global optimization 36, loop identification 37, loop invariant code motion 38, loop unrolling 30, register reassociation 31, and instruction scheduling 32. The invention provides a prefetch driver 34 that operates in concert with such known techniques.

The following pertains to the various elements of the low level optimizer shown on FIG. 8.

- Local optimizations include code improving transformations that are applied on a basic block by basic block basis. For purposes of the discussion herein, a basic block corresponds to the longest contiguous sequence of machine instructions without any incoming or outgoing control transfers, excluding function calls. Examples of local optimizations include local common subexpression elimination (CSE), local redundant load elimination, and peephole optimization.
- Global optimizations include code improving transformations that are applied based on analysis that spans across basic block boundaries. Examples include global common sub-expression elimination, dead code elimination, and register promotion that replaces loads and stores with register references
- Loop identification is the process of identifying sections of code that get executed repetitively (typically this is done through interval analysis).
- Loop invariant code motion is the identification of instructions located with a loop that compute the same result on every loop iteration and the re-positioning of such instructions outside the loop body.
- Register allocation and instruction scheduling is the process of assigning hardware registers to symbolic instruction operands and the re-ordering of instructions to minimize run-time pipeline stalls.

FIG. 9 is a block diagram of a prefetch driver according to the invention. In the figure, code within the low level optimizer (numeric designator 24 on FIG. 8) is supplied to the prefetch driver 34. This is shown in the context of the low level optimizer of FIG. 8 by points A and B on the figure, which points are also shown on FIG. 9 to reference the prefetch driver 34 between both of FIGS. 8 and 9. Loop unrolling may be incorporated into the invention and is shown on FIG. 9 as a module that unrolls the loop a selected number of times 92. Loop body analysis 91 is discussed in greater detail below in connection with FIG. 10.

The prefetch driver estimates the prefetch distance 93 and then partitions the memory references occurring in each loop into disjoint equivalence classes 94 based on the symbolic address expression. Address expression are sorted within each equivalence class 95 and prefetch instructions that are necessary for each equivalence class are calculated 96. Finally, the prefetch instructions are generated 97. A stream of code produced by the prefetch driver includes both the low level intermediate representation and prefetches that have been inserted into the intermediate representation in accordance with the invention herein.

The following detailed description pertains to the various modules shown on FIG. 9:

1. Loop body analysis (see FIG. 10, on which the letters G and H correspond to the same letters on FIG. 9):

a. Identify region constants 190. These are pseudo-registers (symbolic instruction register operands) that are only used and not defined in the loop body. For the purposes of prefetching analysis, only integer region constants are of importance.

b. Identify simple basic loop induction variables 191. A simple basic induction variable (BIV) is a pseudo-register whose loop body definitions can all be expressed in the form:

```
BIV=BIV+biv_delta
```

where "biv_delta" is any arithmetic expression involving only pure or region constants. In the example below, the variable "k" is a region constant, and the variable "i" is a BIV, with a single loop body definition whose "biv_delta" term corresponds to (2*k).

```
k = . . .
i = . . .
loop
     i = i + (2 * k)
end_loop
```

The net loop increment for a BIV is the total amount by which the BIV is incremented on every loop iteration.

A BIV is said to have a well-defined loop increment if the total amount by which the BIV is incremented is the same on every loop iteration. The BIV loop increment in this case is simply the sum of the "biv_delta" values associated with each of its loop body definitions.

Note that a BIV with conditional loop body definitions does not have a well-defined loop increment.

c. Compute and linearize address expressions for memory references 192. This involves first identifying the address expression associated with memory references. Typically this is done by recursively tracing back the reaching definitions for the register operands of base-relative and indexed loads and stores that appear in the loop body, and constructing a binary address expression tree, where the internal tree nodes represent simple arithmetic operations (+,−,*) and the leaf nodes represent either a pure constant, a region constant, or a BIV. The traceback terminates unsuccessfully when a non-BIV register operand has multiple reaching definitions or when the address expression can not be expressed as a simple binary expression tree for any other reason. Memory references whose memory address can not be expressed as such a binary expression tree are not further considered for prefetching purposes.

The address expression tree is then linearized, if possible, with respect to a unique BIV, meaning that it is re-written into the form:

```
a_exp * BIV +b_exp
```

where "a_exp" and "b_exp" are themselves arithmetic expressions involving just sum terms each of which is a product involving either literal or region integer constants. The "BIV" term refers to the value of the basic induction variable at the top of the loop entry basic block (the basic block that is the target of the branch representing the back edge of the loop).

An address expression that can be linearized in this manner is considered to be "affine". The "a_exp" term of an affine address expression multiplied by the BIV's net loop increment is also referred to as the memory stride. Also, associated with each such memory reference is a memory data size that can be inferred from the memory reference opcode e.g. a full-word load would be considered to have a data size of 4-bytes.

Memory references with affine address expressions involving a BIV with a well-defined net loop increment that is a compile-time constant and whose "a_exp" term is non-zero are the only memory references that are further analyzed for data prefetching purposes.

In the example below containing indexed references to 4-byte integer arrays, A, B, C, and D,

```
k = . . .
loop
     . . . A[i + 4] . . .
     . . . B[2*i – 2*k + 8] . . .
     . . . C[D[i]] . . .
     i = i + 1
     end_loop
```

the variable "i" is a BIV and the address expressions associated with the references to A, B, and D would be considered affine, since their memory addresses can be expressed as:

(4) * i+(16+&A[0])

(8) *i+(32–8*k+&B[0])

(4) *i+(&D[0])

respectively, where the notation, "&X[0]" refers to the region constant variable that represents the address of the zero'th element of array X.

Note, however that the address expression associated with the reference to array C is not considered affine.

2. Unroll the loop if possible:

a. Compute maximum prefetch unroll factor U. The objective here is to determine the largest unroll factor U that can be used to minimize prefetch instruction overhead without causing memory strides that are less than or equal to the data cache line size to exceed the data cache line size.

The maximum prefetch unroll factor, U is computed as follows:

U=loop unroll factor computed by other criteria (e.g. loop body size, expected trip count, trip count divisibility etc.)

For each affine address expression reference associated with a BIV with a well-defined constant net loop increment "net_loop_delta" do

```
{
     memory_stride = a_exp * net_loop_delta
     if (memory_stride is a compile-time constant &&
          ABS(memory_stride) <= cache_line_size)
     {
          u = cache_line_size/(ABS(memory_stride))
          U = minimum (U, u)
```

-continued

```
        }
    }
```

b. If U>1, then unroll loop body U times and repeat loop analysis (steps a-c) on unrolled loop body.

3. Estimate the minimum required prefetch iteration distance (PFID):

The prefetch iteration distance is the number of loop iterations in advance that data should be prefetched to have the data be available in the cache when it is needed by the processor, assuming the data was not in the cache to begin with. The PFID is computed based on the expected cache miss latency and the minimum resource-constrained latency for each loop iteration as follows:

PFID=ceiling (avg_miss_latency/avg_loop_iteration_latency)

There are two competing constraints with regard to the PFID choice:

First, the PFID should be sufficiently large to hide the expected average cache miss latency.

Secondly, the PFID should not be so large that the prefetched data is displaced from the cache by an intervening colliding memory reference before it is actually referenced.

It is difficult to determine the optimum average expected memory latency. While the best-case round-trip memory access latency on one system may be, for example about 50 cycles, it is likely to be different on another system that uses a slower bus. Furthermore, bus contention and memory bank conflicts tend to increase the memory access latency.

Nevertheless, the average miss latency is heuristically estimated as the minimum number of processor cycles that elapse between the time a request is sent by the processor to the data cache and the time the data is forwarded to the processor, assuming the data was not present in the cache.

Estimating the average loop iteration latency is even harder to do, even for single-basic block loops. Until scheduling and register allocation are performed, it is not possible to know for sure how many cycles a loop iteration is going to take. Because it is expensive and difficult to compute the achievable loop iteration latency precisely, a lower bound on the achievable loop iteration latency based on machine resource usage is computed instead. This is quite effective for superscalar processors that execute instructions out-of-order and are able to overlap operation latencies at run-time. Typically for such machines, instruction retirement bandwidth constrains the execution cycle count the most. Thus, by focusing on the retirement bandwidth requirements of the instructions present in the loop body, a lower bound on the achievable loop iteration latency can be computed.

Certain instructions that are likely to be eventually deleted should be ignored in computing the loop iteration latency estimate. These may include register-to-register move instructions, subscript instructions that may be eliminated by reassociation, and floating-point multiplies and adds that may be fused into floating-point multiply-and-accumulate instructions.

For instance, suppose a target out-of-order processor can retire two memory instructions and two ALU or floating-point operations per cycle and suppose the loop body code consists of:

5 memory operations,

6 ALU operations and

7 floating-point operations

and that three of the ALU operations participate in addressing expressions that are likely to be eliminated through register reassociation. The lower-bound on the loop iteration latency would then be 5 cycles computed as the larger of 5/2 and ((6-3)+7)/2.

Now, it is also necessary to address the issue of loops that have internal branches. The minimum loop iteration latency for such loops is estimated by using previously collected execution profile information, which indicates the execution count for each basic block in the loop body. The minimum cycle count for each basic block is computed based on the retirement constraints for the instruction mix within the basic block.

The minimum cycle count is summed over each basic block that is executed more than half as many times as the loop entry node to yield an estimate for the minimum loop iteration latency.

4. Identify equivalence classes:

To decide the sort of explicit prefetch instructions to insert into the loop body, uniformly generated equivalence classes of memory references are first identified. These are basically disjoint sets of memory references whose address expressions are known to differ by a compile-time constant. This is done to help clearly detect group spatial and group temporal locality among the different memory references, which in turn can help reduce the prefetch instruction overhead.

Place each affine address expressions associated with a BIV with a compile-time constant net loop increment in a distinct group such that all address expressions within a group share the following properties:

they are all associated with the same BIV

they all have the same "a_exp" term

their "b_exp" terms differ by a compile-time constant

The following algorithm is used to do this:

```
- let the set of uniformly generated equivalence classes, UGEC = { }
- add each affine address expression E(biv,a_exp,b_exp), to a work
list W.
repeat
{
    - remove an address expression Ei(biv, a_exp, b_exp) from the work
    list, W

    - compute the memory stride M for Ei as (a_exp * net biv loop
    increment)

    - if M is not a compile-time constant then substitute some fixed
    constant, C for each non-compile-time constant pseudo-register P,
    that occurs in M and compute the constant-folded memory stride M'

    Also, replace with C, occurrences in "b_exp" of any non-compile-
    time constant pseudo-register P that occurs in M, and constant-fold
    "b_exp" to yield b_exp'

    - if M is a compile-time constant, then let M' = M and b_exp' =
    b_exp

    for each existing equivalence class Q in UGEC
    {
        - choose any representative address expression
        Er(biv, a_exp, b_exp') belonging to Q

        - if biv and a_exp of Er and Ei are not identical, move on to
        the next equivalence class

        - symbolically subtract the b_exp' expression for Ei from Er to
        obtain S

        - if S is a non-zero compile-time constant, add Ei to equivalence
        class Q and move on to consider next address expression on the
```

-continued

```
        work list W
    }

    - if Ei was not added to any existing equivalence class, then add a
    new equivalence class X to UGEC and add Ei to X. Also associate M'
    with the newly created equivalence class X

    } until work list W is empty
```

Consider each equivalence class, Q, in turn and do the following:

5. Sort the address expressions within each equivalence class based on their b_exp' terms and replace multiple address expressions with identical b_exp' terms with a single representative address expression.

Since by construction, the address expressions belonging to the same equivalence class differ in their b_exp'terms by a simple constant, it should always be possible to sort them based on increasing b_exp' values.

Let "E_low" be the address expression in Q with the lowest b_exp' value. Compute a relative equivalence class offset, "eq_offset" for each address expression E in Q, as:

E.eq_offset=E.b_exp'−E_low. b_exp'

6. Compute prefetch instructions needed to ensure full cache miss coverage for equivalence class Q.

The goal here is to insert the fewest number of prefetch instructions in the loop body to ensure that in the steady-state, a prefetch is issued for every distinct cache line referenced by the address expressions in the equivalence class Q. Unnecessary prefetches are avoided if possible by exploiting any group-spatial or group-temporal locality that may be apparent among the memory references within each equivalence class.

The method of determining the fewest number of prefetch instructions needed to ensure full cache miss coverage depends on the magnitude of the memory stride, M', associated with the equivalence class.

If M' is <=cache line size, then a prefetch strategy suited to small strides is employed, otherwise a prefetch strategy suited to large strides is used. Note that for large strides, cache line alignment of data elements needs to be considered.

In either case, it is first necessary to identify clusters of references within the uniformly generated equivalence class. A cluster consists of one or more memory references that occur consecutively in the equivalence class list sorted on "eq_offset", with a well-defined cluster leader. The cluster leader is used to generate prefetch data on behalf of all members of the cluster. The objective here is to weed out those refs within an equivalence class that trail other refs within the equivalence class. The refs that are still left standing are essentially cluster leaders.

The manner in which memory references are grouped into clusters depends on the relative size of the memory stride as compared to the cache line size.

a. Cluster identification for small stride equivalence classes.

It is necessary to consider the address expressions in the equivalence-class in the order of increasing "eq_offset" values and determine whether each address expression trails the very next address expression in the equivalence class and if so, drop it from the equivalence class.

Let B(i) and B(i+1) be adjacent memory refs within the sorted equivalence class list. When the memory stride is <=cache line size, B(i) is considered to be in the same cluster as B(i+1), and therefore omitted for prefetch consideration iff

| B(i+1).eq_offset−B(i).eq_offset |<=prefetch memory distance

where the prefetch memory distance is computed as the product of PFID and the effective memory stride, M' for the equivalence class.

The logic behind this is that if B(i+1) leads a reference B(i) by less than the prefetch memory distance, then there is no real point in inserting a prefetch instruction on behalf of B(i). While some of the initial PFID executions of B(i) within the loop may suffer cache misses, subsequent executions of B(i) would either find its data in the cache or have to wait much less than a full cache miss latency for its data to be retrieved from main memory, since B(i+1) or a prefetch associated with B(i+1) would have initiated the memory retrieval earlier in time. [This is of course assuming that conflict/capacity misses haven't displaced the data from the cache by the time B(i) catches up with B(i+1)].

The last PFID loop iterations can be peeled as described in Mowry et al to avoid the overhead of redundant prefetch instructions that would be executed for data elements not accessed by the original loop.

As shown on FIG. 9, the module that computes the prefetch instructions necessary to determine equivalence class **96** is identified by the letters C and D, which letters are used to indicate a more detailed explanation of the module, which is shown on FIG. 11. In the figure, the prefetch driver is shown to comprise a module computes the prefetches that are needed **96**, where small stride prefetch candidates **201** and large stride prefetch candidates **202** are identified in accordance with a detection module **200**. The algorithm for cluster identification with small strides is given below:

for each address expression "p" in the current equivalence class in sorted order, except the very last address expression

```
{
    - let q be the next address expression in the equivalence class

    - if (ABS(q.eq_offset − p.eq_offset) <= ABS(M' * PFID))
    {
        remove p from equivalence class list
    }
    else
    {
        - mark p as a leader of a cluster
        - let p.trailing_offset = p.leading_offset = p.eq_offset
    }
}
- mark the very last address expression P in the equivalence class list
as a cluster leader
- let p.trailing_offset = p.leading_offset = p.eq_offset
```

The address expressions remaining in the equivalence class after this weeding out process are all cluster leaders.

b. Cluster identification for large stride equivalence classes.

The algorithm for detecting the fewest number of prefetch candidates needed for an equivalence class with a large stride is unfortunately a bit more complicated than the one used for small- stride equivalence classes. The primary reason for this is that with large memory strides, the relative cache line alignment of the memory refs becomes important. For instance, consider the following "C" loop nest:

```
int A[100][100];
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
    {
        . . . A[j][i] . . .
```

-continued

```
        . . . A[j][i+1] . . .
    }
```

The above source code fragment strides through the array A in large increments for each iteration of the inner j-loop. It must be determined whether it is sufficient to insert only one prefetch instruction on behalf of A[j][i+1], with the assumption that the A[j][i] reference is a trailing reference. The answer is no because the two references could straddle a cache line boundary. If this were the case, then the references to A[j][i] could miss the cache, possibly on every iteration of the j-loop, even though data is prefetched for the A[j][i+1] reference.

However, this is not to say that there is no hope of sharing prefetch instructions among references within a uniformly generated equivalence class with a large stride. For instance, if the first reference in the j-loop above had been to A[j-1][i+l] instead of A[j][i], clearly one prefetch instruction would be sufficient for both references.

To make the system immune from the vagaries of relative cache line alignment of references within an equivalence class, yet at the same time exploit obvious temporal locality among the references, a two-pass strategy is used. This strategy is shown in FIG. 12. In the first pass, it is necessary to identify clusters of adjacent references within the current equivalence-class, that are sorted based on their eq_offsets **205**. The distinguishing feature of each such cluster is that the references within the cluster share group spatial locality but no group temporal locality.

The leading reference within each such cluster is responsible for prefetching data both for itself and every other reference in the cluster. To accommodate bad breaks with cache line alignments, a cluster leader may give rise to multiple prefetch instructions, each spaced a cache line apart from the next, until the entire span of the cluster is accounted for. To better explain this, consider the following simple loop, which is possibly the result of the loop unrolling step, where "A" is a double-precision array variable, i.e. w/8-byte elements:

```
i_loop:
    A[i] =
    A[i+1] =
    A[i+2] =
    A[i+3] =
    A[i+4] =
    A[i+5] =
    A[i+6] =
    A[i+7] =
    i = i + 8
end_i_loop;
```

First of all, because the loop BIV, "i", has a net loop increment of eight, and the element size of "A" is 8-bytes, this is a large stride equivalence class, assuming a 32-byte cache line size (8×8 bytes=64 bytes)>32 bytes.

All eight references to "A" are placed into the same cluster because they exhibit group spatial locality, and no group temporal locality. The cluster leader is the reference to A[i+7], and the span of the cluster is 64-bytes (i.e. &A[i+7]–&A[i]). If the prefetch memory distance was computed earlier to be 128-bytes, i.e. corresponding to a prefetch iteration distance of two, it is only necessary to insert three prefetch instructions to account for the entire span of this 8-member cluster. These three prefetches essentially prefetch the following array elements:

```
prefetch A[i+0+16] ;    p/f for cluster stragglers
prefetch A[i+3+16] ;    p/f for cluster stragglers
prefetch A[i+7+16] ;    p/f for cluster leader
```

Regardless of the cache line alignment of the cluster leader, these three prefetch instructions ensure that all the cluster members have memory transactions initiated for data that they reference two iterations in advance. To simplify the actual generation of these types of prefetch instructions, cluster stragglers are removed from the equivalence class right away, and the cluster span for the representative cluster leader is recorded.

b.i. The high level algorithm for the first pass is shown below:

let q=last address expression in the current equivalence class

mark q as a cluster leader

let q.trailing_offset=q.leading_offset=q.eq_offset

for each address expression "p" in the current equivalence class considered in backward sorted order, ignoring "q"

```
{
  - compute distance from current cluster leader, dist_l as follows:
      dist_l = ABS(q.eq_offset − p.eq_offset)
  - compute distance from current cluster trailer, dist_t as follows:
      dist_t = ABS(q.trailing_offset − p.eq_offset)
  - check the following two conditions to determine if p should be
    included in the cluster headed by q:
      a) dist_t <= cache_line_size
      b) dist_l < M'
  if both these conditions are met, then
    - let q.trailer_offset = p.eq_offset
    - remove p from the equivalence class
  else
    - let q = p and mark q as a cluster leader
    - let q.trailing_offset = q.leading_offset = q.eq_offset
```

b.ii. Having identified cluster leaders in the first pass, in the second pass, the algorithm attempts to exploit temporal locality between clusters (see the module identified by numeric designator **206** on FIG. 12). This pass is somewhat similar to the algorithm used to identify prefetch candidates for small-stride equivalence classes. Basically, if two adjacent clusters are less than the prefetch memory distance apart, as measured between the trailing cluster's leader and the leading cluster's trailer, then the trailing cluster may a be removed from further prefetch consideration.

However, rather than simply forgetting about the trailing cluster, it is necessary to merge the trailing cluster with the leading cluster. The algorithm must increase the span of the leading representative cluster because the merged cluster's span is used later on to determine how many prefetch instructions are actually needed. Note that the size of a merged cluster can not be allowed to exceed the effective memory stride M' because otherwise a prefetch instruction is needlessly inserted for the merged cluster's trailing reference. Instead, a merged cluster's size is clamped to be no larger than the effective memory stride.

Another important consideration in deciding to merge clusters is whether the merge would be profitable. In general, for a cluster C whose span is "C.s" bytes, "C.p" prefetch instructions are inserted. "C.p" can be computed as:

$$C.p=[\text{ceiling}\ (C.s/L)]+1$$

where "L" is the cache line size. If a cluster C has "C.b" references within it, then unless (C.p <=C.b),there is no benefit from the locality present within the cluster. This profitability criterion is indirectly applied both in the construction of clusters in the first pass, as well as in the merging of clusters in the second pass. The profitability criterion as applied to the first pass suggests that no two adjacent references within a cluster should be greater than a cache line size apart because, otherwise, it would be better to break the cluster into two sub-clusters. Furthermore, the span of the cluster, as in the distance between the candidate cluster trailer and the current cluster leader, must not become larger than the effective memory stride M', as explained before. A merged cluster's size is therefore clamped to be no greater than the effective stride for this reason.

In the second pass, when deciding whether to merge clusters, the number of prefetches that would be needed for the merged cluster using the above formula is compared to the sum of the prefetches that would be needed if the clusters are not merged.

One subtlety to the cluster merging pass, is that it may sometimes seem unprofitable to merge a cluster C(i) with the next leading cluster C(i+1), even though in a larger context it would have been profitable to merge C(i) with C(i+2), assuming that C(i+2) and C(i) are less than the prefetch memory distance apart. Thus, non-adjacent clusters are examined for merging purposes, and the best cluster to merge with is selected.

If it is decided to merge a cluster C(i) with another cluster C(j), the merged cluster leader's relative offset may become larger than the relative offset of C(j)'s leading reference. Thus, the span of a merged cluster may be larger than simply the difference in the original relative offsets of the cluster leader & cluster trailer. The algorithm used for the second, i.e. cluster merging, pass which is used to exploit temporal locality between clusters is explained below:

The algorithm scans the remaining address expressions in the equivalence class, each of which is a cluster leader identified in the first pass, linearly forwards and looks for merging opportunities with the other leading clusters. In the presentation below, it is assumed for simplicity that the uniformly generated equivalence class has a positive stride.

A necessary condition for a trailing cluster C(i) to be eligible for being merged with a leading cluster C(j) is that:

a) [C(j).trailing_offset−C(i).leading_offset]<pf_memory_distance.

Otherwise, C(j) would be leading cluster C(i) by more than is desired.

Now, let:

m=ceiling [(C(j).trailing._offset−C(i).leading_offset) / M']

where "trailing_offset" and "leading_offset" refer to the eq_offset of the cluster trailer and leader respectively. Note that the trailing cluster may be the result of a previous cluster merger and so it "trailing_offset" may not correspond to a memory reference that actually appears in the source code.

Clearly, it is required that 0<m<PFID.

It is also necessary to define the span of a cluster C(x) to be:

C(x).s=C(x).leading_offset−C(x).trailing_offset

As mentioned earlier, the number of prefetch instructions needed for a distinct cluster C(x) is given by, C(x).p:

C(x).p=[ceiling (C(x).s/L)]+1

where L is the cache line size.

If cluster C(i) is merged with cluster C(j), then the span of the merged cluster C(j') is given by:

```
C(j').s = MAX {M', C(j').leading_offset − C(j').trailing_offset}
    (the MAX operation clamps the merged cluster size at M')
where C(j').leading_offset =
    MAX {C(j).leading_offset, (C(i).leading_offset + (m * M'))}
and C(j').trailing_offset =
    MIN {C(j).trailing_offset, (C(i).trailing_offset + (m * M'))}
```

For the merger of cluster C(i) into cluster C(j) to be profitable in terms of reducing the overall number of prefetches required, the following is necessary:

C(j').p<=(C(i).p+C(j).p),

which means that:

ceiling (C(j').s/L)<=(ceiling (C(i).s/L)+ceiling (C(j).s/L)).

Let the savings accruing from merging cluster C(i) with C(j) into a combined cluster C(j') be defined as:

merger_savings (i,j,j')=(C(i).p+C(j).p)−(C(j').p.

The criterion "b" for cluster merging may now be articulated. For a cluster C(i), among the leading clusters, C(j) that satisfy criterion "a", the system chooses to merge cluster C(i) with one of those clusters, C(j) for which:

merger_savings (i,j,j') is maximized and non-negative.

Note that while the value of "m" computed as mentioned above represents the minimum positive integral multiple of the stride required to achieve an overlap of C(i) with C(j), it is also necessary to check whether using (m−1) could yield a smaller span, and hence fewer prefetch instructions for the merged cluster. Although projecting the cluster C(i) ahead by (m−1) iterations definitely does not cause it to overlap with C(j), the projected C(i) leader may get very close to the C(j) trailer, causing the system to expend fewer prefetches than if C(i) is projected ahead by one more iteration. This can be especially true if the stride is much larger than the individual cluster spans.

Thus, the algorithm scans the remaining address expressions, which represent clusters identified in the first pass, in sorted order and merge trailing clusters with a selected leading cluster by projecting the trailing cluster ahead by either m or (m−1) iterations and checking if both criteria "a" and "b" apply. To project a trailing cluster C(i) ahead m iterations to evaluate whether it should be merged with a leading cluster C(j), the system computes the tentative "leading_offset" and "trailing_offset" values for the proposed merged cluster C(j'), thusly:

```
let C(j').leading_offset = C(i).leading_offset + (m * M')
let C(j').trailing_offset = C(i).trailing_offset + (m * M')
if(M' > 0)
{
C(j').leading_offset = MAX (C(j').leading_offset, C(j).leading_offset)
C(j').trailing_offset = MIN (C(j').trailing_offset, C(j).trailing_offset)
} else
{
C(j').leading_offset = MIN (C(j').leading_offset, C(j).leading_offset)
C(j').trailing_offset = MAX (C(j').trailing_offset, C(j).trailing_offset)
}

adjust C(j').trailing_offset if needed to ensure C(j').s does not exceed M'
as follows:
```

-continued

```
C(j').s = ABS (C(j').leading_offset - C(j').trailing_offset)

if (C(j').s > ABS(M'))
{
    C(j').trailing_offset = C(j').leading_offset - M'
    C(j').s = ABS(M')
}
```

When a cluster C(i) is successfully merged with a cluster C(j) into a cluster C(j'), C(i) is removed from the equivalence class list.

7. Generate the prefetch instructions required for each remaining cluster leader.

It is necessary to consider each cluster leader in turn, and where "trailing_offset" is different than "leading_offset" for any cluster leader, insert as many prefetches as needed to cover the cluster's entire span, i.e. from "leading_offset" down to "trailing_offset", each prefetch instruction address spaced L bytes apart.

More specifically, if the memory reference corresponding to a cluster leader is represented by the instruction:

```
load disp(Rb),Rt
```

where "disp" is a displacement value and Rb and Rt are pseudo-registers corresponding to the base register and target register of the load, then one or more prefetch instructions are inserted into the code stream as follows:

```
prefetch_inst new_disp (Rb)
```

where "new_disp" is computed as disp +(M*PFID) +pf_disp, where "pf_disp" represents the constant offset that needs to the added to the memory address referenced by the cluster leader, to form the base address from which it is necessary to prefetch ahead by the prefetch memory distance. The algorithm used to emit the prefetch instructions is given below:

```
- let L = cache line size for each remaining cluster C(i) in the equivalence
class
{
    - let disp = displacement of memory reference instruction associated
    with the leader address expression for cluster C(i)
    - if(C(i).leading_offset == C(i).trailing_offset) then
    {
        pf_disp = 0
        - emit prefetch_inst with new_disp = disp + (M * PFID)
    }
    else {
        if (M > 0)
        {
            - let cur_offset = C(i).leading_offset
            - let final_offset = C(i).trailing_offset
        }else
        {
            - let cur_offset = C(i).trailing_offset
            - let final_offset = C(i).leading_offset
        }
        - let pf_disp = cur_offset - C(i).eq_offset
        while (cur_offset > final_offset) do
        {
            - emit prefetch_inst with new_disp = disp +
            (M*PFID) + pf_disp
            - let cur_offset = cur_offset - L
            - let pf_disp = pf_disp - L
        }
        - emit one final prefetch to the account for the final member
        of the cluster (i.e. the "leader" for a negative memory stride,
```

-continued

```
        else the "trailer") with new_disp = disp + (M*PFID) +
        (final_offset - C(i).eq_offset)
        }
}
```

Note that in computing new_disp, the original memory stride value M, computed as (a_exp * net biv loop increment), is used and not the constant folded value M'. This may require materializing a run-time region constant expression in a register, outside the loop body, and inserting an explicit add instruction within the loop body to form the prefetch instruction address thusly:

```
Rm = (a_exp * net_biv_loop_increment) * PFID
loop
    Rx = Rm + Rb
    prefetch_inst new_disp'(Rx)
    load disp(Rb),Rt
end_loop
```

where new_disp'=disp +pf_disp. If the prefetch instruction supports an addressing mode which causes the effective memory address to be computed as the sum of two register values, then the add operation may be omitted by folding in the new_disp' value in Rm outside the loop body, yielding Rm', and specifying the Rm and Rb registers operands of the prefetch instruction directly, as shown below:

```
Rm' = ((a_exp * net_biv_loop_increment) * PFID) + new_disp'
loop
    prefetch_inst Rm'(Rb)
    load disp(Rb),Rt
end_loop
```

However, if "disp" itself is a run-time value, as opposed to a simple constant, then an explicit add operation is unavoidable:

```
Rm' = ((a_exp * net_biv_loop_increment) * PFID) + pf_disp
loop
    Rx = Rb + disp
    prefetch_inst Rm'(Rx)
    load disp(Rb),Rt
    end_loop
```

and if the prefetch instruction does not support a register+register addressing mode, then two add operations may be needed:

```
Rm = (a_exp * net_biv_loop_increment) * PFID
loop
    Rx1 = Rb + disp
    Rx2 = Rx1 + Rm
    prefetch_inst pf_disp(Rx2)
    load disp(Rb),Rt
end_loop
```

Note however, that these new add instructions may be eliminated through register reassociation. In fact, the prefetch instruction(s) and the beneficiary memory reference instruction may be able to share the same base register through register reassociation, allowing the add instructions to be deleted:

```
Rp - initialized to the address of the memory location referenced by the
     load on the first loop iteration
Rm = (a_exp * net_biv_loop_increment)
Rdelta = (Rm * PFID) + pf_disp
loop
     prefetch_inst Rdelta(Rp)
     load 0(Rp),Rt
     Rp = Rp + Rm
end_loop
```

Furthermore, if the target architecture supports an auto-increment addressing mode (e.g. PA-RISC, IBM Power PC), then the increment of the new base register Rp may be folded into the load instruction itself.

In terms of the code placement of the prefetch instruction itself, to start with, the prefetch instruction(s) may be placed adjacent to the beneficiary memory reference instruction. Subsequently, the instruction scheduling phase may re-order the prefetch instruction(s) as needed to improve performance. In doing this, memory dependencies between the prefetch instruction and other memory references in the loop body may be ignored and assuming the prefetch instruction is guaranteed not to raise an exception, it may be freely scheduled across basic blocks as well.

Although the invention is described herein with reference to the preferred embodiment, one skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention. Accordingly, the invention should only be limited by the Claims included below.

I claim:

1. A compiler, comprising:

means in a low level optimizer for analyzing and efficiently inserting explicit data prefetch instructions into loops of applications;

subscript expression analysis means for determining data prefetching requirements;

means for recognizing cache line reuse patterns across loop iterations to eliminate unnecessary prefetch instructions; and

means for limiting insertion of explicit prefetch instructions to situations where a lower bound on an achievable loop iteration latency is unlikely to be increased as a result of said prefetch instruction insertion.

2. The compiler of claim 1, wherein analysis and explicit data cache prefetch instruction insertion are performed by said compiler in a machine instruction level optimizer.

3. The compiler of claim 1, further comprising:

means for exploiting execution profiles from previous runs of an application during insertion of prefetch instructions into innermost loops with internal control flow.

4. The compiler of claim 1, wherein said prefetch insertion means is integrated with other low-level optimization phases.

5. The compiler of claim 4, said other low-level optimization phases comprising:

any of loop Unrolling, register reassociation, and instruction scheduling.

6. A method for mitigating or eliminating cache misses in a low level optimizer, comprising the steps of:

performing loop body analysis;

unrolling loops to reduce prefetch instruction overhead;

identifying uniformly generated equivalence classes of memory references in a code stream, where said

equivalence classes represent disjoint sets of memory references occurring in a loop whose address expressions can be expressed as a linear function of the same basic loop induction variable and are known to differ only by a compile time constant, allowing the detection of group spatial and group temporal locality among said different memory references;

computing an effective memory stride for each of the equivalence classes;

determining the number of prefetch instructions needed for full cache miss coverage for each equivalence class, where the number of prefetch instructions that needs to be inserted is a function of the style of prefetching desired, including dumb prefetching that inserts an explicit prefetch instruction for each memory reference, baseline prefetching that inserts as many prefetch instructions as possible without affecting the resource minimum loop iteration latency, and selective prefetching that inserts as many prefetch instructions as are required to ensure full cache miss coverage, exploiting any group-spatial or group-temporal locality that may be apparent among memory references within a uniformly generated equivalence class; and

inserting prefetch instructions identified into said code stream.

7. The method of claim 6, further comprising the step of:

estimating a prefetch iteration distance for a loop as the ratio of average miss latency and average loop iteration latency, where the average loop iteration latency is derived from a resource-constrained lower bound on a cycle count based on machine resource usage.

8. The method of claim 6, further comprising the step of:

substituting a fixed constant value for unknown terms into the address expressions for memory references to runtime dimensioned arrays to facilitate partitioning of such references into disjoint equivalence classes.

9. The method of claim 6, further comprising the step of:

determining the number of prefetch instructions that are needed for each uniformly generated equivalence class for a selective prefetching strategy.

10. The method of claim 9, further comprising the step of:

sorting the address expressions for memory references belonging to an equivalence class based on their relative constant differences.

11. The method of claim 9, further comprising the step of:

determining an effective memory stride for the memory references associated with each equivalence class and classifying the effective memory stride as being either large or small based on whether it is greater than the cache line size.

12. The method of claim 9, further comprising the step of:

determining prefetch memory distance for the memory references associated with each equivalence class as the product of effective memory stride and prefetch iteration distance for the loop.

13. The method of claim 9, further comprising the step of:

removing memory references within a small-stride equivalence class that trail other memory references within said equivalence class by less than the prefetch memory distance, wherein memory references that remain are cluster leaders.

14. The method of claim 9, further comprising the step of:

grouping the memory references belonging to a large-stride equivalence class that are sorted by their constant address expression differences into clusters each of which has a distinct memory reference designated as the cluster leader and zero or more memory references designated as cluster trailers.

15. The method of claim 9, further comprising the step of:

merging clusters represented by their leaders to profitably exploit group temporal locality in a pairwise fashion.

16. The method of claim 6, further comprising the steps of:

deciding which equivalence classes to insert prefetch instructions for an under the baseline prefetching strategy by first sorting uniformly generated equivalence classes based on a prefetch cost/expected benefit criteria, and only committing to insert prefetch instructions for those equivalence classes with the best cost/expected benefit ratio, without causing resource-based minimum loop iteration latency to be exceeded.

17. The method of claim 6, further comprising the steps of:

running through clusters in each equivalence class;

generating explicit prefetch instructions for each cluster; and

inserting said prefetch instructions into the code stream.

* * * * *

US005797013A

United States Patent [19]
Mahadevan et al.

[11] **Patent Number: 5,797,013**
[45] **Date of Patent: Aug. 18, 1998**

[54] **INTELLIGENT LOOP UNROLLING**

[75] Inventors: **Uma Mahadevan**, Sunnyvale; **Lacky Shah**, Fremont, both of Calif.

[73] Assignee: **Hewlett-Packard Company**, Palo Alto, Calif.

[21] Appl. No.: **564,514**

[22] Filed: **Nov. 29, 1995**

[51] **Int. Cl.**^{6} **G06F 9/45**
[52] **U.S. Cl.** **395/709**; 395/588
[58] **Field of Search** 395/705, 709, 395/588, 580

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,265,253	11/1993	Yamada	395/700
5,367,651	11/1994	Smith et al.	395/700
5,386,562	1/1995	Jain et al.	395/650

OTHER PUBLICATIONS

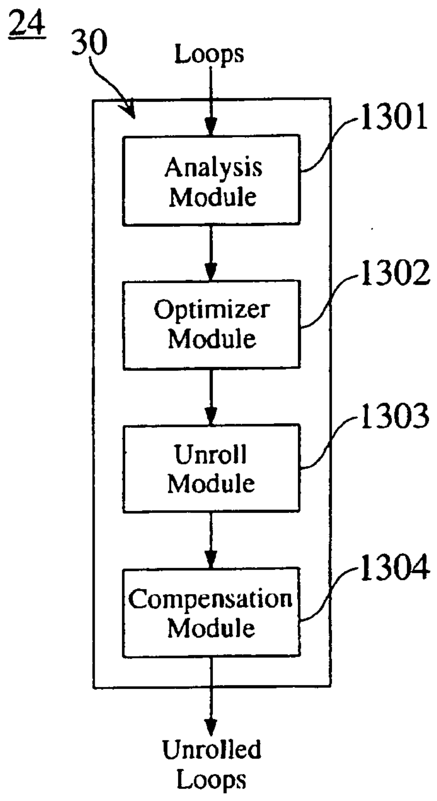
"A Comparative Evaluation of Software Techniques to Hide Memory Latency", John et al., Proc. of the 28th Ann. Hawaii Int'l Conf., 1995, pp. 229–238.
"Schedule driven Loop Unrolling for Parallel Processors", System Sciences, 1991 Annual Hawaii Int'l Conference, 1991, vol. II pp. 458–467.
"Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler", Davidson et al., Dept of Comp. Science, Univ. of Va. pp. 1–14.
"Unrolling Loops in Fortran," Dongarra et al., Soft. Practice and Experience, vol. 9, 1979, pp. 219–226.
Hendren et al., "Designing Programming Languages for the Analyzability of Pointer Data Structures," Comput. Lang., vol. 19, No. 2, pp. 119–134 (1993).
Weiss et al., "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," ACM, pp. 105–109 (1987).

Primary Examiner—Emanuel Todd Voeltz
Assistant Examiner—Kakali Chaki

[57] **ABSTRACT**

A compiler facilitates efficient unrolling of loops and enables the elimination of extra branches from the loops, including the elimination of conditional branches from unrolled loops with early exits. Unrolling also enhances other optimizations, such as prefetch, scalar replacement, and instruction scheduling. The unroll factor is calculated to determine the amount of loop expansion and the optimum location to place compensation code to complete the original loop count, i.e. before or after the unrolled loop. The compiler is applicable, for example, to modern RISC architectures, where the latency of memory references and branches is higher than that of integer and floating point arithmetic instructions.

16 Claims, 13 Drawing Sheets



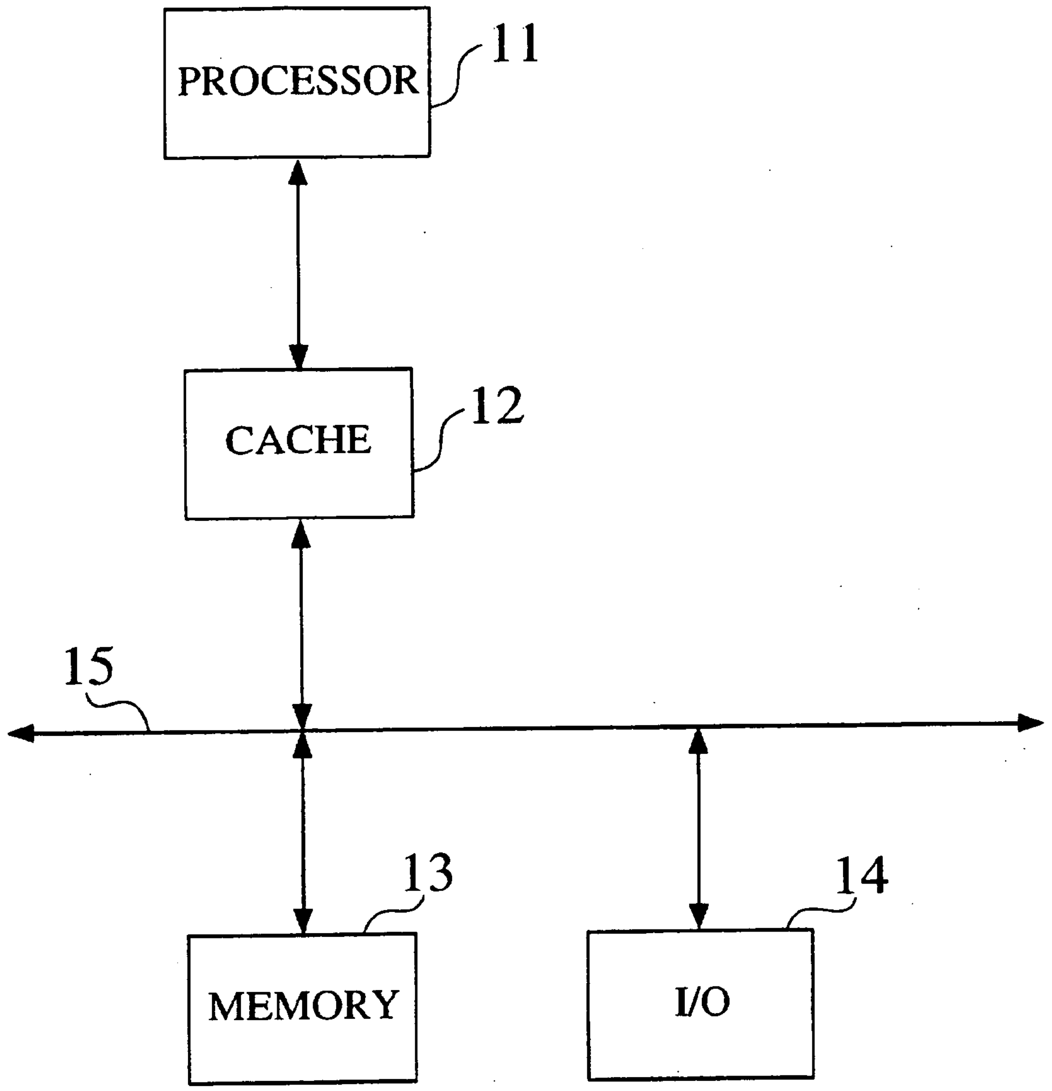


Fig. 1 (PRIOR ART)

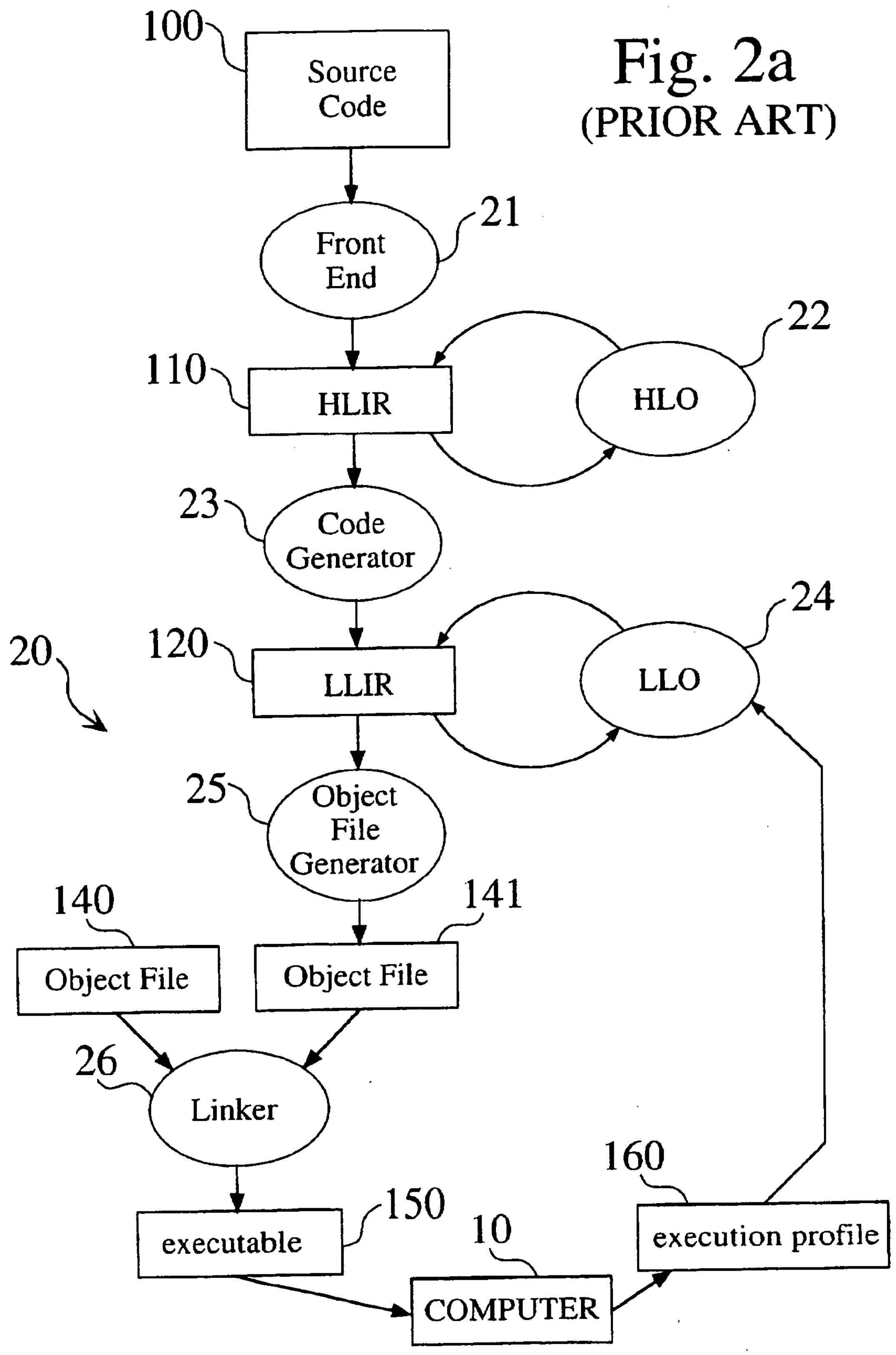


Fig. 2a
(PRIOR ART)

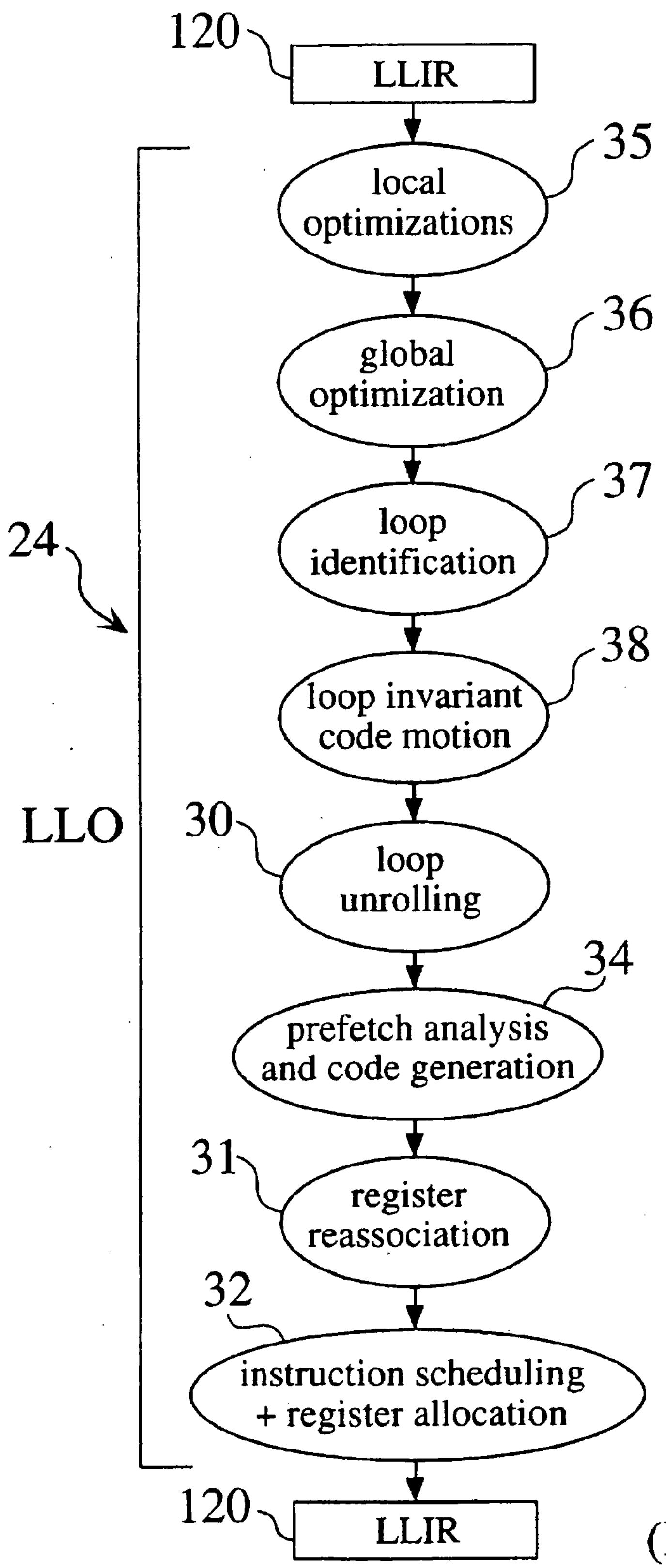
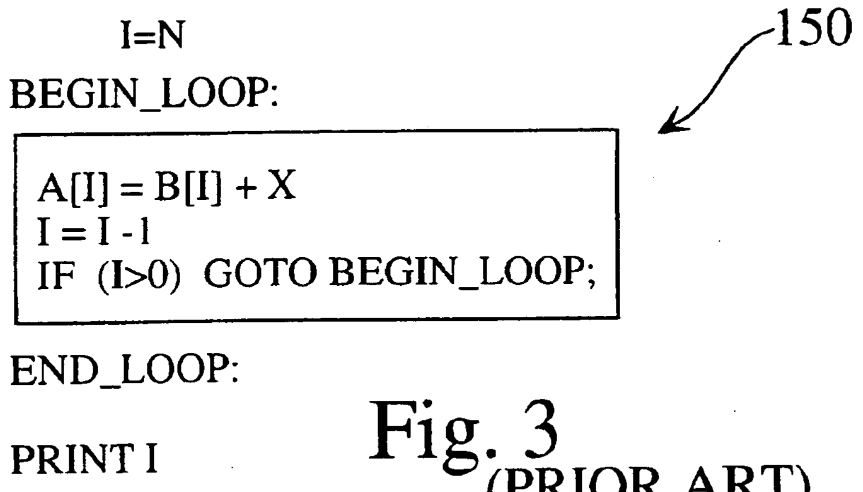
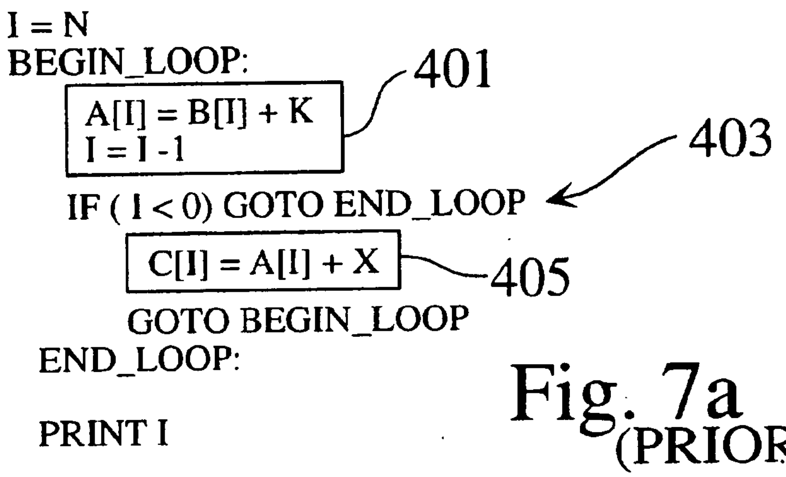


Fig. 2b
(PRIOR ART)



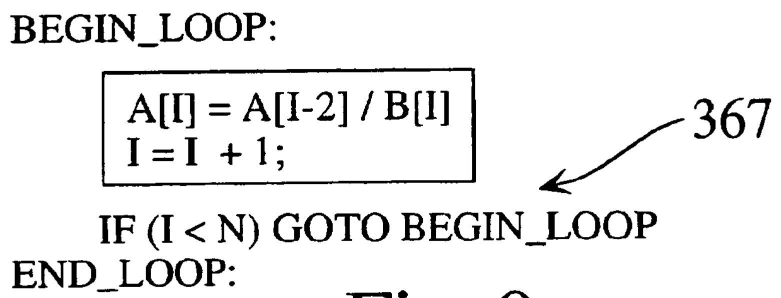
```
      I=N
BEGIN_LOOP:
  A[I] = B[I] + X
  I = I -1
  IF  (I>0)  GOTO BEGIN_LOOP;
END_LOOP:
PRINT I
```

Fig. 3 (PRIOR ART)



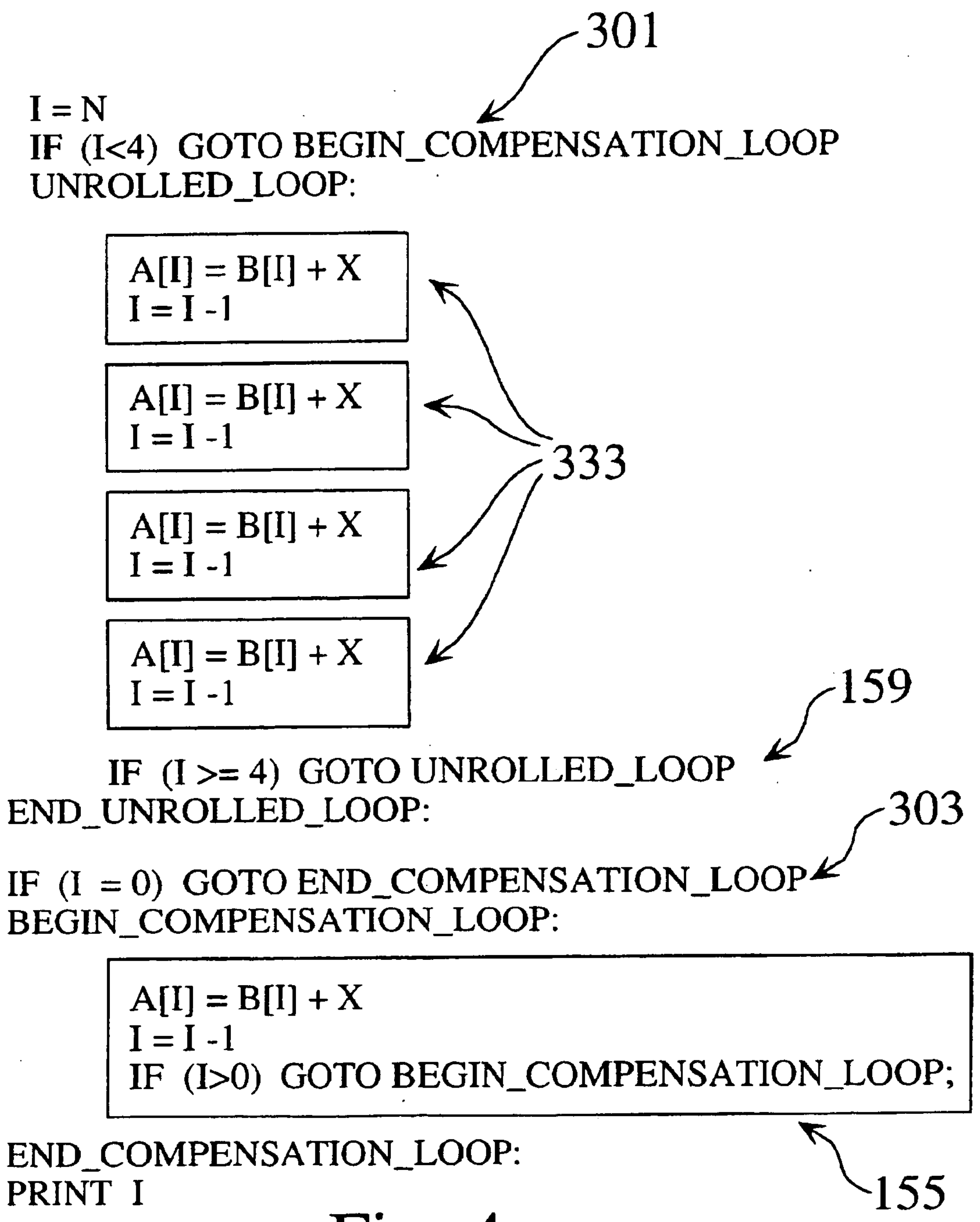
```
I = N
BEGIN_LOOP:
    A[I] = B[I] + K
    I = I -1
    IF ( I < 0) GOTO END_LOOP
        C[I] = A[I] + X
        GOTO BEGIN_LOOP
  END_LOOP:

  PRINT I
```

Fig. 7a (PRIOR ART)



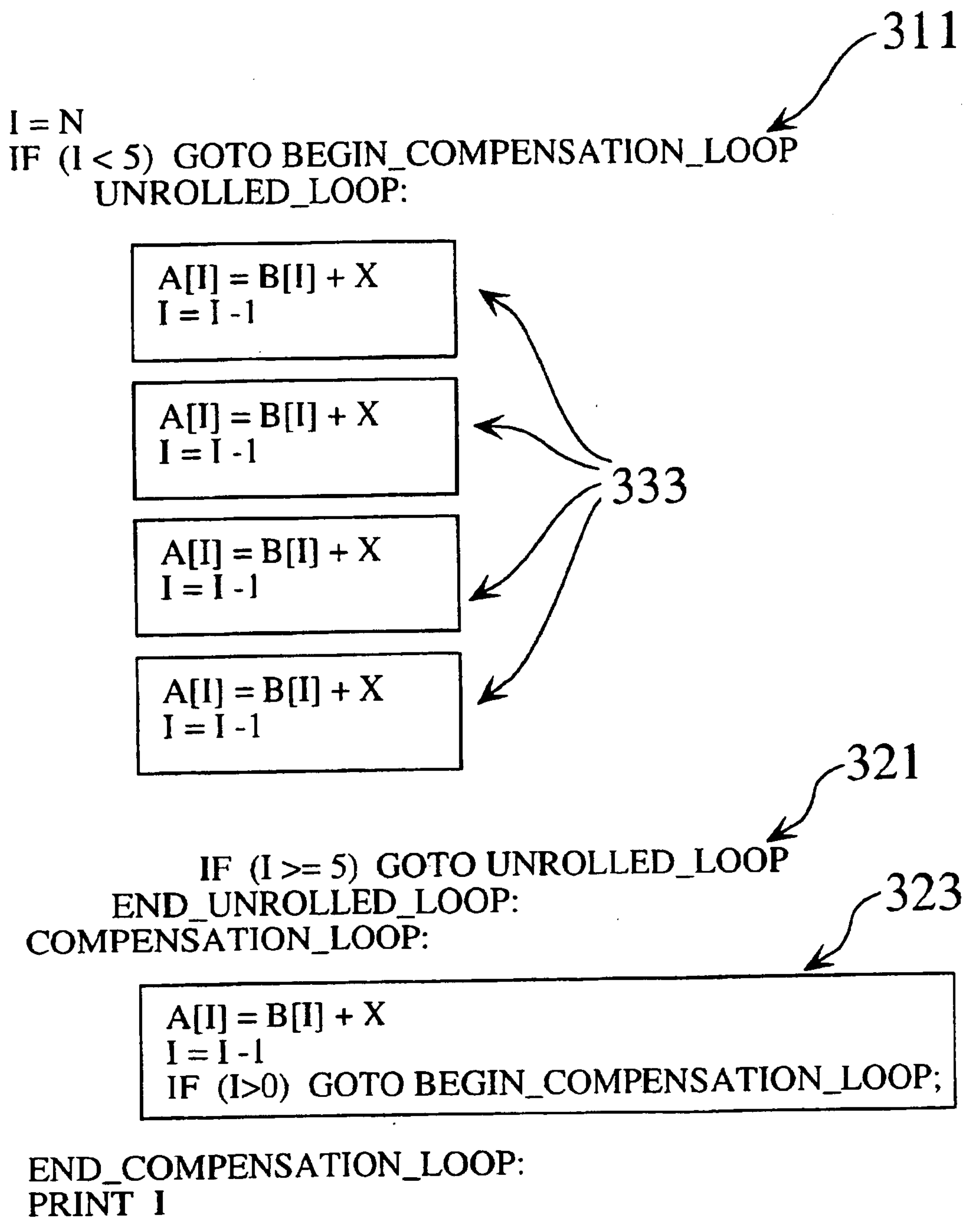
```
BEGIN_LOOP:
      A[I] = A[I-2] / B[I]
      I = I  + 1;
    IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 9 (PRIOR ART)



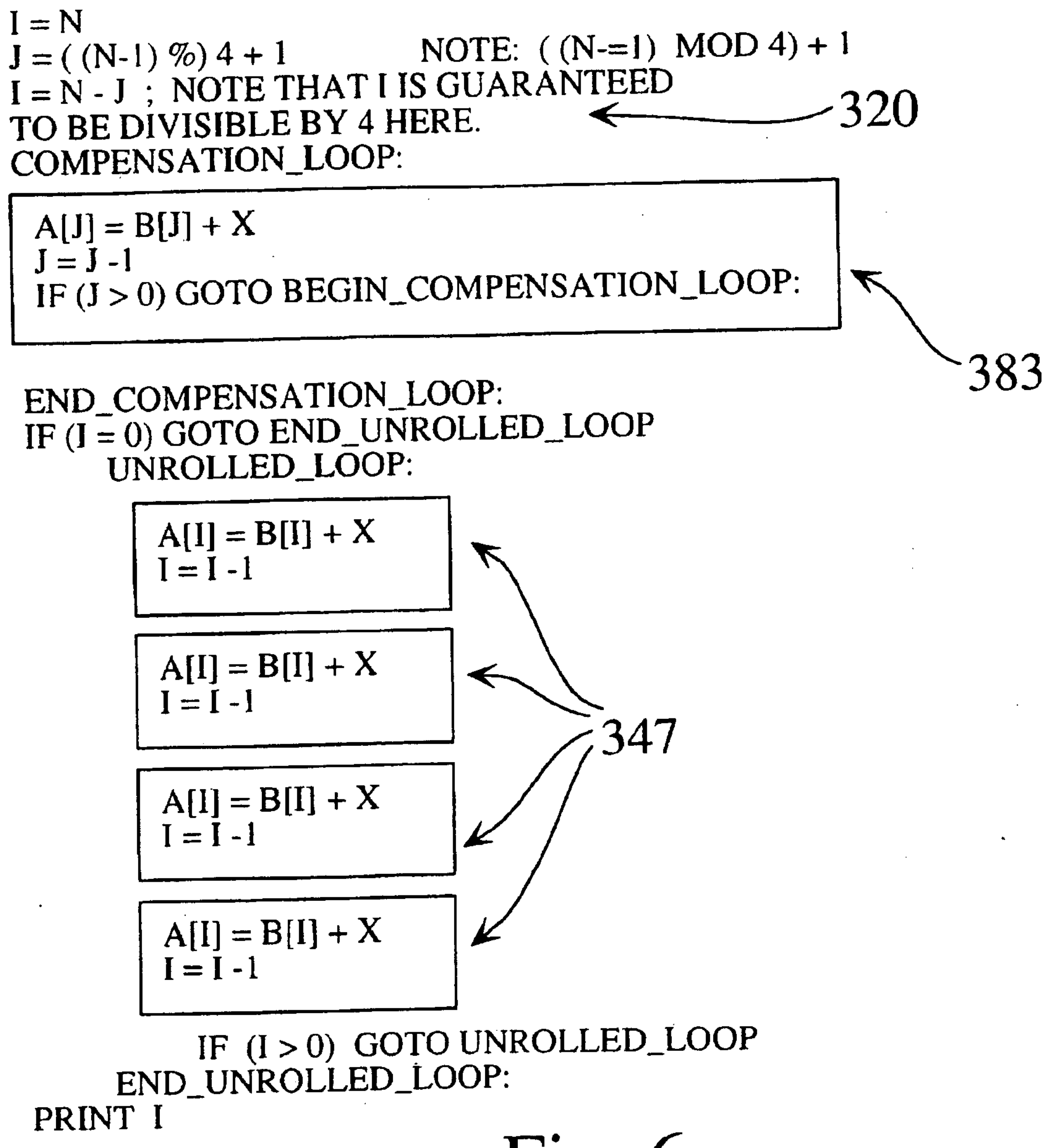
```
I = N
IF (I<4) GOTO BEGIN_COMPENSATION_LOOP
UNROLLED_LOOP:

    A[I] = B[I] + X
    I = I -1

    A[I] = B[I] + X
    I = I -1

    A[I] = B[I] + X
    I = I -1

    A[I] = B[I] + X
    I = I -1

    IF (I >= 4) GOTO UNROLLED_LOOP
END_UNROLLED_LOOP:

IF (I = 0) GOTO END_COMPENSATION_LOOP
BEGIN_COMPENSATION_LOOP:

    A[I] = B[I] + X
    I = I -1
    IF (I>0) GOTO BEGIN_COMPENSATION_LOOP;

END_COMPENSATION_LOOP:
PRINT I
```

Fig. 4
(PRIOR ART)



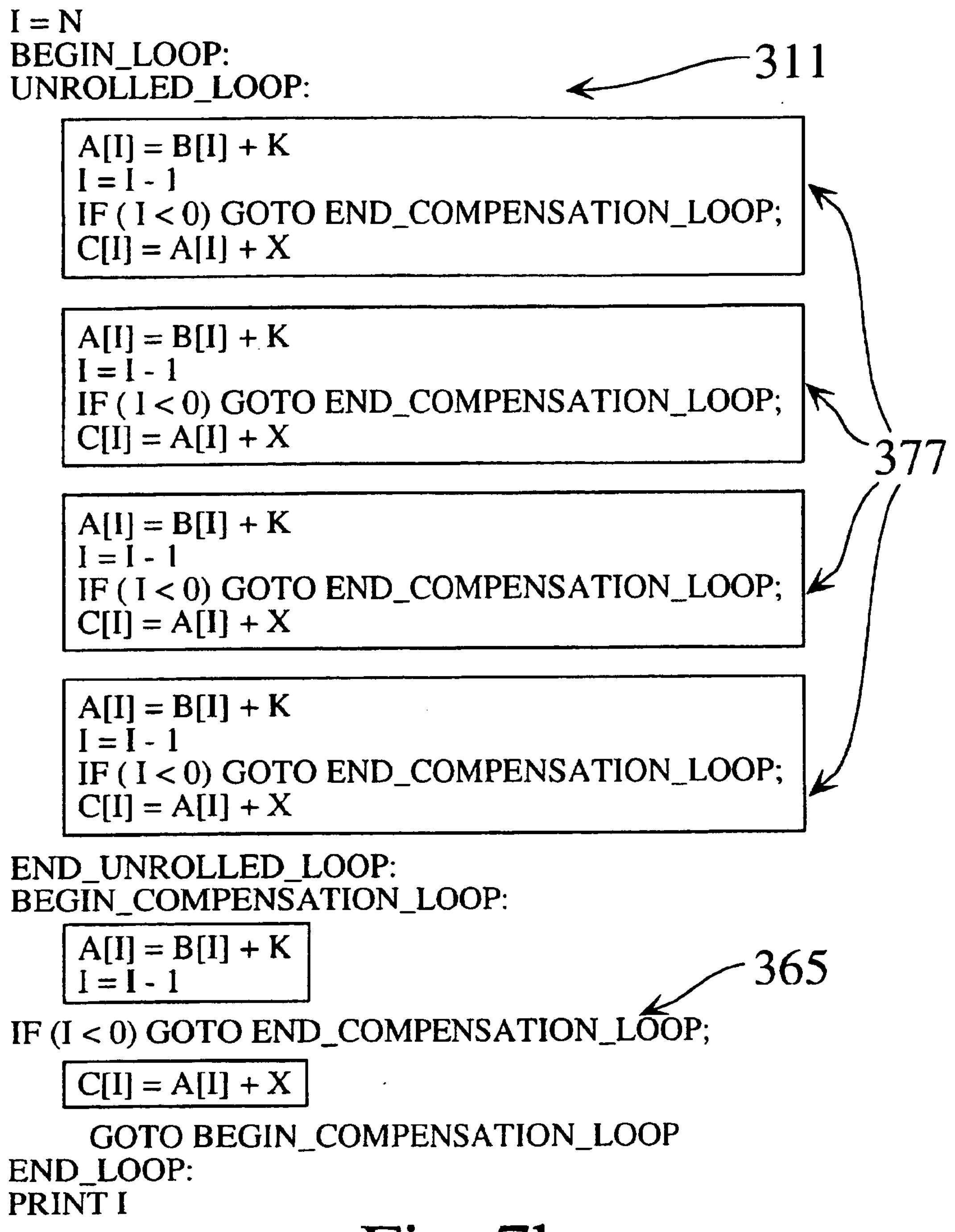
```
I = N
IF (I < 5) GOTO BEGIN_COMPENSATION_LOOP
     UNROLLED_LOOP:

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

            IF (I >= 5) GOTO UNROLLED_LOOP
      END_UNROLLED_LOOP:
 COMPENSATION_LOOP:

          A[I] = B[I] + X
          I = I -1
          IF (I>0) GOTO BEGIN_COMPENSATION_LOOP;

 END_COMPENSATION_LOOP:
 PRINT I
```

Fig. 5



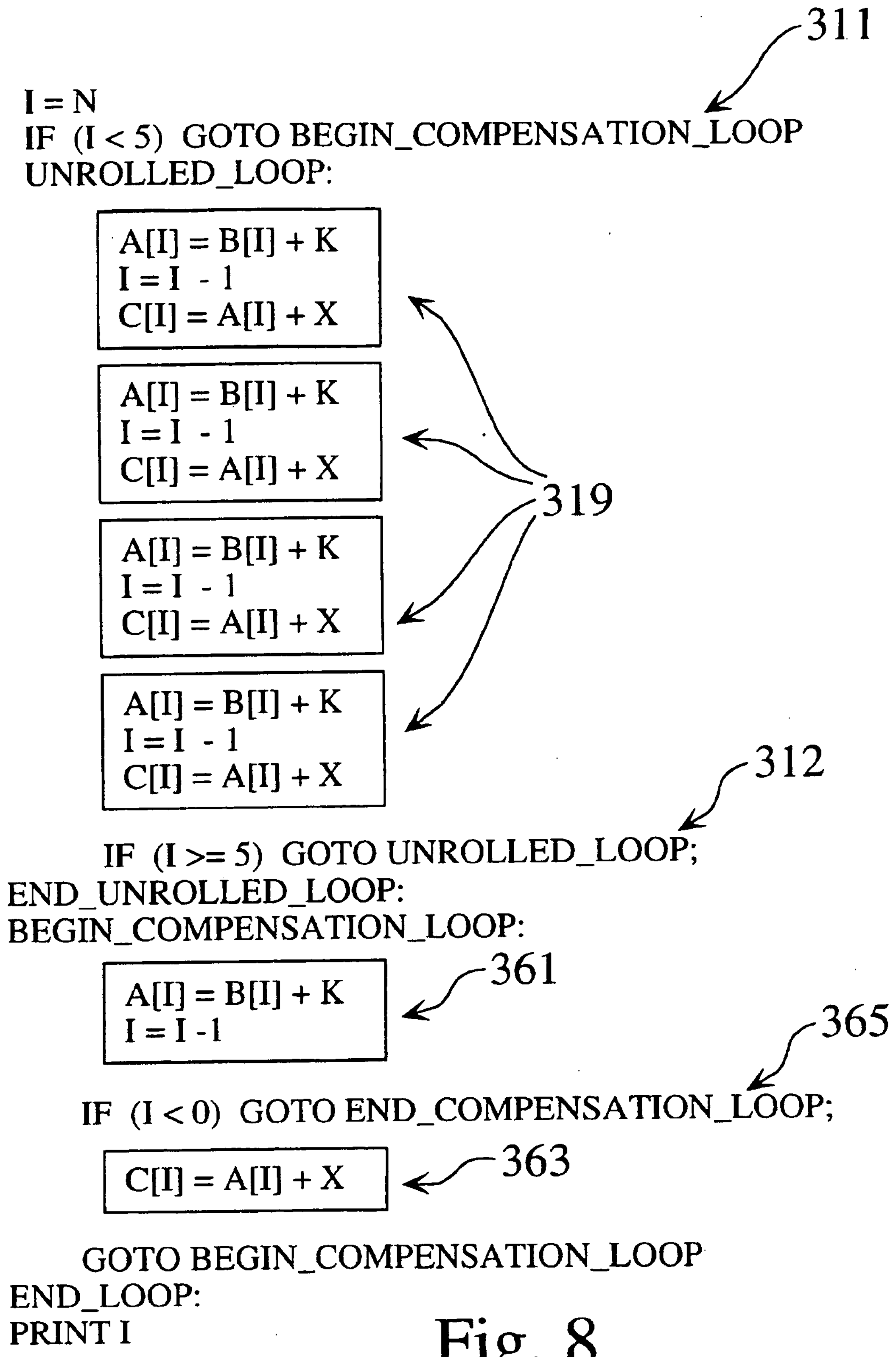
```
I = N
J = ( (N-1) %) 4 + 1            NOTE:  ( (N-=1)  MOD 4) + 1
I = N - J  ;  NOTE THAT I IS GUARANTEED
TO BE DIVISIBLE BY 4 HERE.
COMPENSATION_LOOP:
  A[J] = B[J] + X
  J = J -1
  IF (J > 0) GOTO BEGIN_COMPENSATION_LOOP:
END_COMPENSATION_LOOP:
IF (I = 0) GOTO END_UNROLLED_LOOP
      UNROLLED_LOOP:
            A[I] = B[I] + X
            I = I -1
            A[I] = B[I] + X
            I = I -1
            A[I] = B[I] + X
            I = I -1
            A[I] = B[I] + X
            I = I -1
            IF  (I > 0)  GOTO UNROLLED_LOOP
      END_UNROLLED_LOOP:
PRINT  I
```

Fig. 6



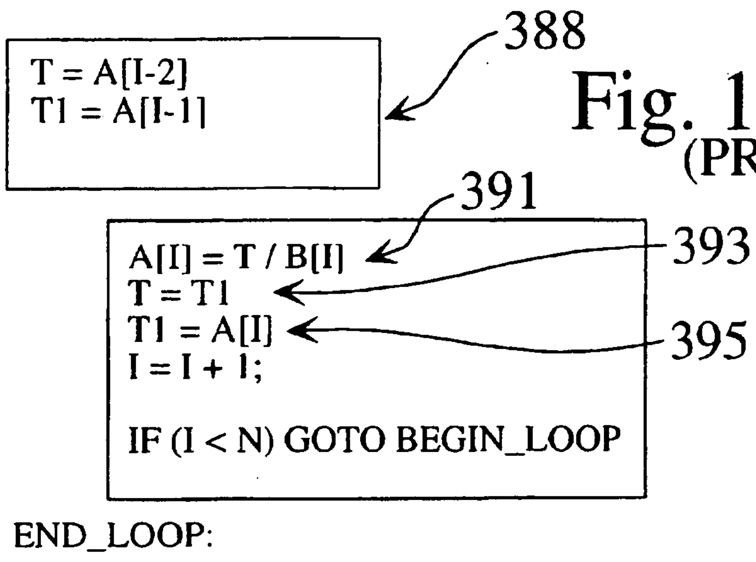
```
I = N
BEGIN_LOOP:
UNROLLED_LOOP:
    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X
END_UNROLLED_LOOP:
BEGIN_COMPENSATION_LOOP:
    A[I] = B[I] + K
    I = I - 1
IF (I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X
      GOTO BEGIN_COMPENSATION_LOOP
END_LOOP:
PRINT I
```

Fig. 7b (PRIOR ART)



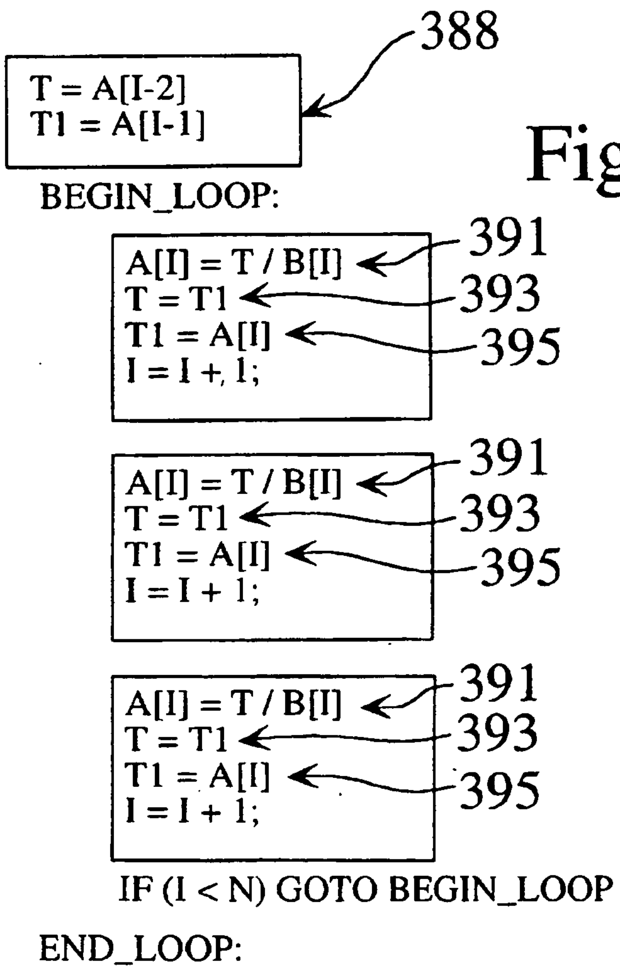
```
I = N
IF (I < 5) GOTO BEGIN_COMPENSATION_LOOP
UNROLLED_LOOP:
        A[I] = B[I] + K
        I = I - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I - 1
        C[I] = A[I] + X

    IF (I >= 5) GOTO UNROLLED_LOOP;
END_UNROLLED_LOOP:
BEGIN_COMPENSATION_LOOP:
        A[I] = B[I] + K
        I = I -1
    IF (I < 0) GOTO END_COMPENSATION_LOOP;
        C[I] = A[I] + X
    GOTO BEGIN_COMPENSATION_LOOP
END_LOOP:
PRINT I
```

Fig. 8



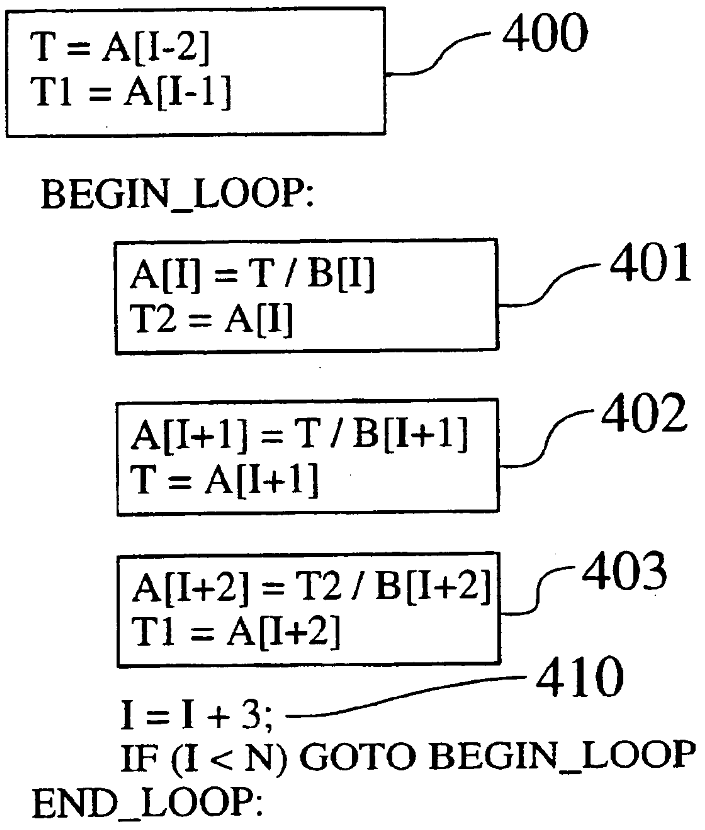
```
T = A[I-2]
T1 = A[I-1]

A[I] = T / B[I]
T = T1
T1 = A[I]
I = I + 1;

IF (I < N) GOTO BEGIN_LOOP

END_LOOP:
```

Fig. 10 (PRIOR ART)



```
T = A[I-2]
T1 = A[I-1]
BEGIN_LOOP:
    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;

    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;

    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;
    IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 11 (PRIOR ART)



```
T = A[I-2]
T1 = A[I-1]

BEGIN_LOOP:

    A[I] = T / B[I]
    T2 = A[I]

    A[I+1] = T / B[I+1]
    T = A[I+1]

    A[I+2] = T2 / B[I+2]
    T1 = A[I+2]

    I = I + 3;
    IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 12

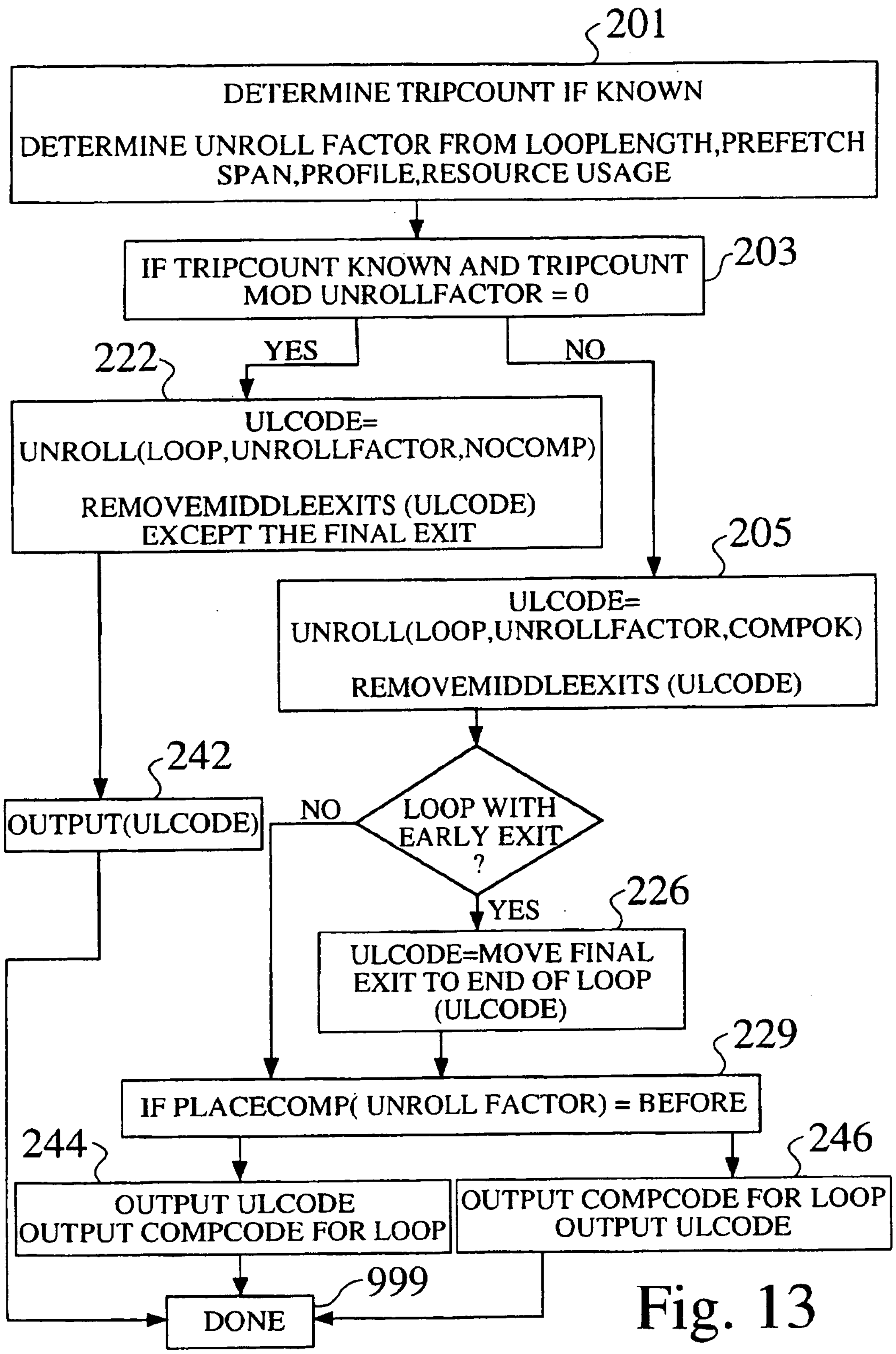

201
DETERMINE TRIPCOUNT IF KNOWN
DETERMINE UNROLL FACTOR FROM LOOPLENGTH,PREFETCH SPAN,PROFILE,RESOURCE USAGE
203
IF TRIPCOUNT KNOWN AND TRIPCOUNT MOD UNROLLFACTOR = 0
YES
NO
222
ULCODE= UNROLL(LOOP,UNROLLFACTOR,NOCOMP)
REMOVEMIDDLEEXITS (ULCODE) EXCEPT THE FINAL EXIT
205
ULCODE= UNROLL(LOOP,UNROLLFACTOR,COMPOK)
REMOVEMIDDLEEXITS (ULCODE)
242
OUTPUT(ULCODE)
NO
LOOP WITH EARLY EXIT ?
YES
226
ULCODE=MOVE FINAL EXIT TO END OF LOOP (ULCODE)
229
IF PLACECOMP(UNROLL FACTOR) = BEFORE
244
OUTPUT ULCODE
OUTPUT COMPCODE FOR LOOP
246
OUTPUT COMPCODE FOR LOOP
OUTPUT ULCODE
999
DONE

Fig. 13

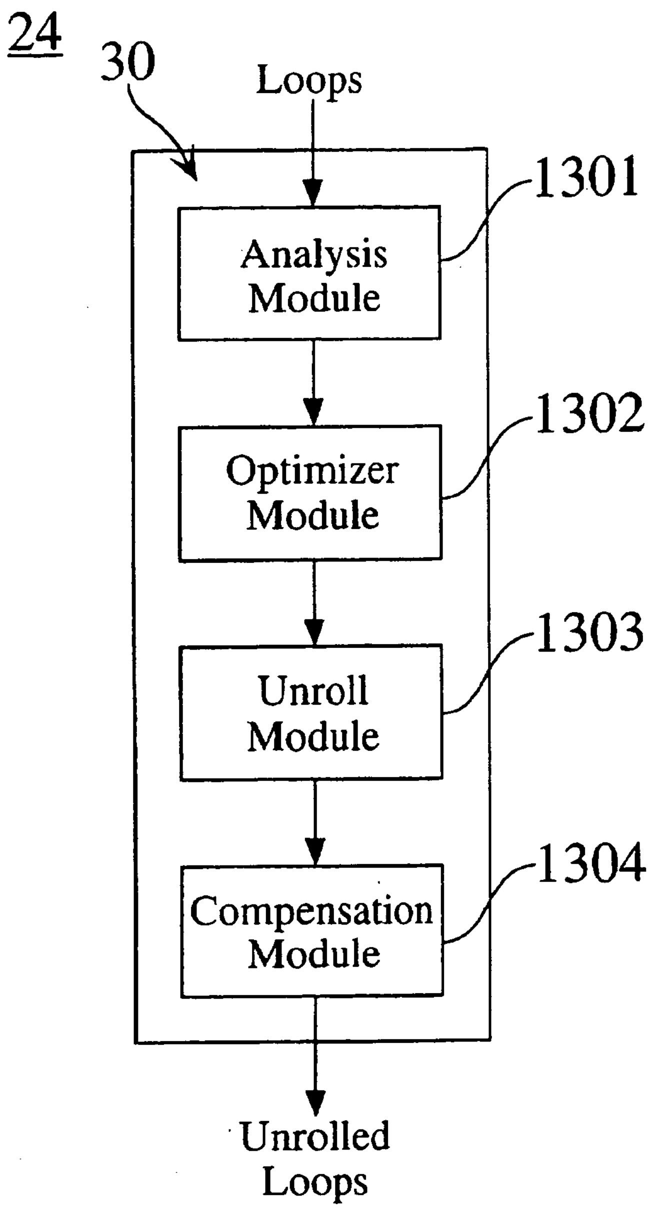

24
30
Loops
Analysis Module
1301
Optimizer Module
1302
Unroll Module
1303
Compensation Module
1304
Unrolled Loops

Fig. 14

INTELLIGENT LOOP UNROLLING

BACKGROUND OF THE INVENTION

1. Technical Field

The invention relates to compilers. More particularly, the invention relates to techniques for unrolling computation loops in a compiler so as to generate code which executes faster.

2. Description of The Prior Art

FIG. 1 is a block schematic diagram of a uniprocessor computer architecture, including a processor cache. In the figure, a processor **11** includes a cache **12** which is in communication with a system bus **15**. A system memory **13** and one or more I/O devices **14** are also in communication with the system bus.

FIG. 2*a* is a block schematic diagram of a software compiler **20**, for example as may be used in connection with the computer architecture shown in FIG. 1. The compiler Front End component **21** reads a source code file (**100**) and translates it into a high level intermediate representation (**110**). A high level optimizer **22** optimizes the high level intermediate representation **110** into a more efficient form. A code generator **23** translates the optimized high level intermediate representation to a low level intermediate representation (**120**). The low-level optimizer **24** converts the low level intermediate representation (**120**) into a more efficient (machine-executable) form. Finally, an object file generator **25** writes out the optimized low-level intermediate representation into an object file (**141**). The object file (**141**) is processed along with other object files (**140**) by a linker **26** to produce an executable file (**150**), which can be run on the computer **10**. In the invention described herein, it is assumed that the executable file (**150**) can be instrumented by the compiler (**20**) and linker (**26**) so that when it is run on the computer **10**, an execution profile (**160**) may be generated, which can then be used by the low level optimizer **24** to better optimize the low-level intermediate representation (**120**). The compiler **20** is discussed in greater detail below.

The compiler is the piece of software that translates source code, such as C, BASIC, or FORTRAN, into a binary image that actually runs on a machine. Typically the compiler consists of multiple distinct phases, as discussed above in connection with FIG. 2*a*. One phase is referred to as the front end, and is responsible for checking the syntactic correctness of the source code. If the compiler is a C compiler, it is necessary to make sure that the code is legal C code. There is also a code generation phase, and the interface between the front-end and the code generator is a high level intermediate representation. The high level intermediate representation is a more refined series of instructions that need to be carried out. For instance, a loop might be coded at the source level as:

for(I=0,I<10,I=I+1),

which might in fact be broken down into a series of steps, e.g. each time through the loop, first load up I and check it against 10 to decide whether to execute the next iteration.

A code generator takes this high level intermediate representation and transforms it into a low level intermediate representation. This is closer to the actual instructions that the computer understands. An optimizer component of a compiler must preserve the program semantics (i.e. the meaning of the instructions that are translated from source code to an high level intermediate representation, and thence to a low level intermediate representation and ultimately an executable file), but rewrites or transforms the code in a way that allows the computer to execute an equivalent set of instructions in less time.

Modern compilers are structured with a high level optimizer (HLO) that typically operates on a high level intermediate representation and substitutes in its place a more efficient high level intermediate representation of a particular program that is typically shorter. For example, an HLO might eliminate redundant computations. With the low level optimizer (LLO), the objectives are the same as the HLO, except that the LLO operates on a representation of the program that is much closer to what the machine actually understands.

FIG. 2*b* is a block diagram showing a low level optimizer for a compiler, including a loop unrolling component **30** according to the invention. The low level optimizer **24** may include any combination of known optimization techniques, such as those that provide for local optimization **35**, global optimization **36**, loop identification **37**, loop invariant code motion **38**, prefetch **34**, register reassociation **31**, and instruction scheduling **32**.

Source programs translated into machine code by compilers consists of loops, e.g. DO loops, FOR loops, and WHILE loops. Optimizing the compilation of such loops can have a major effect on the run time performance of the program generated by the compiler. In some cases, a significant amount of time is spent doing such bookkeeping functions as loop iteration and branching, as opposed to the computations that are performed within the loop itself. These loops often implement scientific applications that manipulate large arrays and data instructions, and run on high speed processors.

This is particularly true on modern processors, such as RISC architecture machines. The design of these processors is such that in general the arithmetic operations operate a lot faster than memory fetch operations. This mismatch between processor and memory speed is a very significant factor in limiting the performance of microprocessors. Also, branch instructions, both conditional and unconditional, have an increasing effect on the performance of programs. This is because most modern architectures are super-pipelined and have some sort of a branch prediction algorithm implemented. The aggressive pipelining makes the branch misprediction penalty very high. Arithmetic instructions are interregister instructions that can execute quickly, while the branch instructions, because of mispredictions, and memory instructions such as loads and stores, because of slower memory speeds, can take a longer time to execute.

Modern compilers perform code optimization. Code optimization consists of several operations that improve the speed and size of the complied code, while maintaining semantic equivalence. Common optimizations include:

- prefetching data so that they are available in cache memory when needed;
- detecting calculations as computing constants and performing the calculation at compile time;
- scalar replacement which keeps the value of a variable in a register within the loop;
- moving calculations outside of loops where possible; and
- performing code scheduling, which consists of rearranging the order of and modifying instructions to achieve faster running but semantically equivalent code.

Many modern compilers also employ an optimizing technique known as loop unrolling to generate faster running code. In its essence, loop unrolling takes the inner loop, i.e. the code between the beginning and the end of the loop, and repeats it in the inner loop some number of times, e.g. four times. It then executes the unrolled loop one-fourth as many times as it would have executed the original loop. The number of times the loop is replicated within the unrolled loop is called the unroll factor. Because the number of times the original loop is executed is not always divisible by the unroll factor, a compensation loop code often has to be generated to execute the remaining of instructions of the original loop that are not executed by the unrolled loop.

As discussed above, such loops as DO, FOR and WHILE loops are common in programs, especially in scientific and other time-consuming programs. Frequently 80% of the running time of a program can be in a few small loops. As a result anything that can speed up such loops is of great value in making a more efficient compiler.

Consider the simple loop shown on FIG. 3. The three instructions in the inner loop 150 are executed N times. According to the prior art, this loop can be unrolled with an unroll factor of four to produce the code shown on FIG. 4, where the inner loop (less the exit condition) is replicated 4 times 333. This loop is also followed by a test 303 to see if the full original loop has been completed, and a compensation loop 155 which is executed to the complete the original loop trip count if it has not been completed.

An inspection of the loop shows that it is semantically equivalent to the loop of FIG. 3 because the same function is performed and the same result is achieved. However, the loop of FIG. 4 runs much faster for several reasons. First, the conditional branch 159 which exits the loop is executed only once for each time through the unrolled loop rather than once for each time through the original loop. Assuming an unroll factor of four as is shown here, this saves ¾ of a conditional branch per original loop iteration. More importantly, other compiler optimizations interact with loop unrolling and are able to do a much better job of optimizing the unrolled loop, such as that identified by numeric designator 333, as compared to the original loop, identified by numeric designator 150. In an unrolled loop, there are more operations that could be scheduled in parallel, more opportunity to do scalar replacement and other optimizations, and more possibilities to do prefetching.

The tests identified by numeric designators 301 and 303 are also of interest. These are the conditional branches which have a higher probability of being mispredicted. Anything that can be done to eliminate one of them will be very useful.

Prior art loop unrolling techniques have certain disadvantages. For example, J. J. Dongara and A. R. Hinds, *Unrolling Loops in FORTRAN*, describes how one can unroll loops manually by duplicating code. This is an early solution to optimizing code that it is not even implemented by the compiler.

S. Weiss and J. E. Smith, *A Study of scalar compilation techniques for pipeline supercomputers*, discuss unrolling in the compiler. Here the authors address the simple situations:

a) Cases where the loop count is known at compile time. They do not address loop unrolling when the loop count is only known at run time.

b) Cases where the loop exit appears only at the beginning or end of the loop. They do not address the situation of unrolling loops with early exits (loops whose exit may occur in the middle of the loop).

The authors do not address the following issues:

a) Determining whether to place the compensation code before or after the unrolled loop.

b) Tuning of the iteration count to reduce branch misprediction.

c) Factors that affect the unroll factor.

L. J. Hendren and G. R. Gao, *Designing Programming Languages for the Analyzability of Pointer Data Structures*, addresses the issue of unrolling loops as part of compiler optimization. They do not however discuss:

a) Unrolling loops with early exits:

b) Tuning of the iteration count to reduce branch misprediction;

c) Factors that affect the unroll factor;

d) Whether to place a compensation code at the beginning or end of the loops; and

e) Compiling loops whose trip count is only known at run time.

J. Davidson, S. Jinturkar, *Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler*, Dept. of Computer Science, Thornton Hall, University of Virginia disclose a code transformation, referred to as aggressive loop unrolling, in a retargetable optimizing compiler where the loop bounds are not known at compile time. Various factors were analyzed to determine how and when loop unrolling should be applied, resulting in an algorithm for loop unrolling in which execution-time counting loops (i.e. a counting loop whose iteration count is not trivially known at compile time) are unrolled and loops having complex control-flow are unrolled. However, they do not discuss:

a) Unrolling loops having early exits;

b) Tuning of the iteration count to reduce branch misprediction

c) Factors that affect the unroll factor; and

d) Whether to place a compensation code at the beginning or end of the loops.

Another part of the loop unrolling prior art is shown on FIG. 7, which illustrates a loop having an early exit (also referred to as a WHILE loop), consisting of an exit test and branch 403 in the middle of the loop between computations 401 and 405. According to the prior art, the code, including the exit 403, is replicated four times in an unrolled loop. The number of branches in the loop are however not reduced with this optimization.

While some of the techniques discussed in the prior art are applicable to compilers for all computers, only some of them are particularly applicable for modern RISC computers, where branch instructions form a lot bigger bottleneck than in earlier technologies. Also compilers for RISC architectures are a lot more aggressive and the interactions of various optimizations plays a key role in the quality of the final code.

SUMMARY OF THE INVENTION

The invention provides a new compiler that can unroll more loops than previous algorithms. It also significantly reduces the number of branch instructions by cleverly handling the iteration count and by converting loops with early exits to regular FOR loops. The invention also provides for computing the unroll factor and the placement of the compensation loop by taking a lot of other optimizations into consideration.

The compiler:

Eliminates time consuming conditional branch instructions from the compensation code loop by replacing the conditional exit of the main unrolled loop to always exit with at least one iteration which has yet to be executed by the compensation code. This eliminates the need to test for zero remaining loops.

Determines whether it is better to place the compensation code at the beginning or the end of the unrolled loop according to which one would likely provide the better optimization. Generally, it prefers to put the compensation loop in front of the main loop if the unroll factor is a power of two and after the main loop if the unroll factor is not a power of two.

Computes the unroll factor by taking into account the interactions of other optimizations like prefetch, scalar replacement and register allocation, and also taking into account hardware features like number of functional units. Unrolling loops over-aggressively or under-aggressively can inhibit other optimizations or make them less effective.

Converts loops with early exit to loops with exit at the end to apply more efficient optimizations to the loop. It does this by ensuring that the compensation code is always executed at least once, enabling the compiler to eliminate the exit tests from the unrolled loop.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a block schematic diagram of a uniprocessor computer architecture including a processor cache;

FIG. 2*a* shows a block schematic diagram of a modern software compiler;

FIG. 2*b* shows schematic diagram of the low level optimizer;

FIG. 3 shows a simple program loop;

FIG. 4 shows the loop of FIG. 3 that has been unrolled four times according to the prior art;

FIG. 5 shows the loop of FIG. 3 unrolled according to the invention and having an eliminated branch;

FIG. 6 shows an unrolled loop having pre-loop compensation code;

FIG. 7*a* shows a simple WHILE loop;

FIG. 7*b* shows the WHILE loop of FIG. 7*a* that has been unrolled according to the prior art;

FIG. 8 shows the WHILE loop of FIG. 7*a* that has been unrolled according to the invention;

FIG. 9 shows a schematic representation of a simple loop;

FIG. 10 shows the loop of FIG. 9 with scalar replacement according to the prior art;

FIG. 11 shows the loop of FIG. 10 unrolled;

FIG. 12 shows a schematic representation of the loop of FIG. 11 after copy elimination;

FIG. 13 shows a schematic representation of the compiler logic which determines the unroll factor, compensation code placement, and other optimizations; and

FIG. 14 is a block schematic diagram of a compiler for a programmable machine in accordance with the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention provides a new compiler that features smart unrolling of loops.

The invention provides a prefetch driver 34 that operates in concert with such known techniques. The following discussion pertains to the various elements of the low level optimizer shown on FIG. 2*b*:

Local optimizations include code improving transformations that are applied on a basic block by basic block basis. For purposes of the discussion herein, a basic block corresponds to the longest contiguous sequence of machine instructions without any incoming or outgoing control transfers, excluding function calls. Examples of local optimizations include local common sub-expression elimination (CSE), local redundant load elimination, and peephole optimization.

Global optimizations include code improving transformations that are applied based on analysis that spans across basic block boundaries. Examples include global common sub-expression elimination, loop invariant code motion, dead code elimination, register allocation and instruction scheduling.

Loop invariant code motion is the identification of instructions located with a loop that compute the same result on every loop iteration and the re-positioning of such instructions outside the loop body.

Register allocation and instruction scheduling is the process of assigning hardware registers to symbolic instruction operands and the re-ordering of instructions to minimize run-time pipeline stalls where the processor must wait on a memory fetch from main memory or wait for the completion of certain complicated instructions that take multiple cycles to execute (eg. divide, square root instructions).

One important phase of the compiler identifies loops and access patterns to estimate how many cycles are devoted to loop iterations. In the invention, the compiler translates the higher level application code into an instruction stream that the processor executes, and in the process of this translation the compiler unrolls loops.

The longer unrolled loops allow the compiler to provide several advantages, such as:

1) It eliminates the extra branch exits. This saves CPU cycles by not having to execute the branch instructions and also helps reduce branch misprediction. Why this is important is evident if one considers most modern RISC architectures. These architectures have a long pipeline that is fed by an instruction fetch mechanism. When the fetch mechanism encounters a branch, it tries to predict if the branch is going to be taken or not. It then fetches instructions based on this prediction. The prediction is necessary to keep the pipeline from stalling. If the architecture's prediction is correct (this is determined when the branch instruction completes execution which is a few cycles after it has been fetched), then everything works fine; else all the instructions that have been fetched after the branch are discarded and the new instructions fetched based on the correct outcome of the branch. This penalty of discarding fetched instructions and fetching new ones, when the branch is mispredicted, is known as the branch misprediction penalty and it is very significant for most modern architectures. It is of the order of 5–10 cycles per branch instruction that is mispredicted. By reducing the branch instructions, the number of branches that get mispredicted automatically reduces.

2) It can better insert prefetches and effect other optimizations into the longer inner loop code. When the loop is unrolled, there are more memory instructions in the loop and also the memory stride (the distance between the memory accesses of an instruction in two consecutive iterations) is bigger. If a loop is unrolled four times, the memory stride goes up by four. This helps the prefetch to do a more effective job. When the memory stride increases, as long as it is less than the cache line size (which is architecture dependent), the prefetches become more effective. When the memory stride becomes greater than the cache line size the prefetches can hurt. Hence the loops should be unrolled such that the memory stride is lesser than the cache line size

whenever prefetch instructions are going to be generated for the loop and whenever possible.

3) Scalar replacement/recurrence elimination inserts copies at the loops to keep the value of a variable live in the next iteration (see, for example FIGS. 9, 10, and 11). These copies can be eliminated by unrolling the loop a certain number of times.

4) Longer sequences of non-branching instructions can achieve an overlap between instructions that have nothing to do with memory and those that do. This is known as instruction scheduling and explained below. While the access time between the processor and the cache is typically 1 to 5 cycles, the retrieval time from cache to memory is often on the order of 10 to 100 cycles. When the processor actually gets to the point where the data item is needed from memory, if the data is not in cache, it might take 100 processor cycles to fetch it from main memory. Where the compiler can optimize the longer inner loop code, it may only be necessary to wait for 20 cycles because 80 cycles worth of look up time is hidden or overlapped with the execution of other instructions.

Loop unrolling is integrated with other low level optimization phases, such as the prefetch insertion algorithm, register reassociation, and instruction scheduling. The new compiler yields significant performance improvements for some industry-standard performance benchmarks, for example on the SPEC92 and SPEC95 benchmarks on the Hewlett-Packard Company (Palo Alto, Calif.) PA-8000 processor.

The following discussion explains compiler operation in the context of a loop within an application program. Loops are readily recognized as a sequence of code that is iteratively executed some number of times. The sequence of such operations is predictable because the same set of operations is repeated for each iteration of the loop. It is common practice in an application program to maintain an index variable for each loop that is provided with an initial value, and that is incremented by a constant amount for each loop iteration until the index variable reaches a final value. The index variable is often used to address elements of arrays that correspond to a regular sequence of memory locations.

In the compiler, it has been found that the low level optimizer component of a compiler is in a good position to deduce the number of cycles required by a stretch of code that is repetitively executed and this information can be used to determine the optimal unroll factor. As discussed above, the concept of loop unrolling is not new, but use of smart unrolling is new. For example, FIG. 4 shows the loop of FIG. 3 after the loop has been unrolled four times. Thus, instead of executing the loop 100 times if N were 4, the loop is executed 25 times.

FIG. 5 shows the output code that is generated by the invention in contrast to the code generated by the prior art, as shown on FIG. 4. The replicated inner loops 333 are the same. Also, the compensation loop 323 is the same as the prior art compensation loop 155 of FIG. 4. However, the loop test at 301 and 159 of FIG. 4 now tests to exit if I>=5 rather than I>=4, as can be seen at 311 and 321 of FIG. 5. The effect is to ensure that the compensation loop is always executed at least once. This eliminates the need to test for the zero case (303 in FIG. 4). This eliminates the branch instruction 303 on FIG. 4. As indicated above, the elimination of this branch instruction significantly increases the speed of the compiled code by reducing the number of branch instructions that get mispredicted.

It is also possible to put the compensation code in front of the main loop, as is shown on FIG. 6. Here the compensation

loop 383 is in front of the repetitive unrolled loops 347. In the general case, putting the compensation code before the main unrolled loop is less efficient than putting afterwards, because calculating the loop trip count requires a remainder operation which involves high latency divide operations.

However, if the unroll factor is a power of two, as in this case where the unroll factor is 4, the remainder calculation is a simple shift operation. Because unroll factors of 2, 4, or 8 are common, the compensation code can be placed in front in front of the unroll loop for negligible cost. As a practical matter, it is often advantageous to put the compensation loop in front of the unrolled loop to benefit from other optimizations such as register reassociation. When the compensation loop is placed before the unrolled loop, the variable that keeps track of the iteration count is not always needed after the unrolled loop. When the compensation loop is placed after the unrolled loop, this variable is always needed after the unrolled loop as there is an exposed use in the compensation loop. This exposed use can inhibit aggressive register reassociation. In the preferred embodiment, the architecture of the computer and the interactions with other optimizations dictate an unroll factor, and if it is a power of two, the compensation code is inserted in front of the unroll loop.

Another optimization technique that is part of the invention herein disclosed rearranges loops with early exits (which are henceforth referred to as WHILE loops). These loops are characterized by the fact that some of the inner loop code is done before the loop test, and some after the loop test as is shown on FIG. 7*a*. Here the loop has an exit branch 403 in the middle with inner loop operations 401 before it, and other inner loop operations 405 after it.

The optimization taught in the prior art for this loop is shown on FIG. 7*b*. Notice that the whole inner loop, including the exit instruction, is replicated 377 four times. This can be improved by converting the unrolled loop into a FOR loop with an exit condition of (unroll factor +1) as opposed to unroll factor (e.g. 5 instead of 4 in this case), as is shown on FIG. 8. This guarantees that the unrolled loop is exited before it would have to exit due to the WHILE condition. Because none of the branches at 377 on FIG. 7*b* are executed, the WHILE exit instruction can be removed, as is shown at 319 on FIG. 8. Thus, there is only one place that there is a WHILE loop exit, i.e. at 365.

The technique herein disclosed ensures that the unrolled loop exits before it would take any of the WHILE loop exits, so that the WHILE test can be removed from the unrolled loop. It is necessary to ensure that the compensation code is always executed at least once.

The following discusses how the unroll factor interacts with the scalar replacement optimization. This is particularly important because the form of this optimization determines the unroll factor. Consider the loop shown on FIG. 9. Notice that the value of A[I] stored in the inner loop at 367 is loaded again two loop iterations later, when the same statement loads A[I-2] with a value of I which is incremented by 2. The idea behind scalar optimization which is well known in the prior art, is to save array values in temporary variables if they are accessed shortly within the next few iterations. Thus, the loop can be modified as is shown on FIG. 10.

Here, the array reference A[I-2] at 391 is replaced with T, and it is followed by two instructions at 393 and 395 which assign values to T and T1. Two scalar temporary variables are necessary because the value of the two most recent array values must be saved. The value of A[I-1] would have been stored in the previous iteration in T1 and that is going to be used in the next iteration. We move T1 to T and T will be accessed in the next iteration. Similarly T1, to which A[I] is assigned will be moved to T in the next iteration and used 2 iterations from now.

The instruction at 395 appears to make an indexed reference to the array A and that suggests that an array access must be made to get the number to put into T1, which would be a high latency operation and would lose all that was gained by the optimization. What actually happens is the optimizer recognizes that the value of A1 is stored two instructions earlier at 391, and that A[I] is resident in a register which can be stored into T1 without accessing A[I]. As T1 is likely to be assigned to a register, this operation is a register to register instruction.

The foregoing illustrates how the various optimization techniques are interrelated allowing the loop unrolling optimizer to generate code which is clearly not optimal in itself but is optimized by other optimizers in the compiler. Initialization code is inserted at 388 to define initial values of T and T1.

FIG. 11 shows how such a loop can be unrolled according to the prior art if an unroll factor of three had been chosen. The prior art does not specify the selection of an unroll factor of three here, but a factor of three, or a multiple of three is optimal because it allows other parts of the optimizer to generate the code shown on FIG. 12. The selection of an unroll factor here of three or a multiple of three is important because three values of the array must be kept A[I], A[I-1] and A[I-2] if the array references are to be avoided.

Three variables T, T1 and T2 are used, making it possible for other well known code optimization techniques to generate code, such as that shown on FIG. 12. This eliminates shuttling the temporary data from T to T1. This is an example of where the nature of the code in the loop and its effect on scalar optimization forces a particular unroll factor. In general, one lists the various indexes in the loops and sorts them to notice the maximum distance between them.

In the above case the distance between the index I and I-2, is 2. Adding one to this value computes a primary unroll factor, which is three in this case. This is an acceptable unroll factor. However, if it turned out to be very small, one might want to multiply it by a constant to get a larger unroll factor. Alternatively, if the primary unroll factor was very large, one might want to divide it by the loop increment of it was a number other than one. The reasons for selecting a particular unroll factor are discussed below.

Determining the unroll factor.

Classically the prior art uses a standard unroll factor for all loops. Typically the number used is four. In the invention, the unroll factor is calculated for each loop depending on various factors. At one extreme some loops are not unrolled at all, and other loops are unrolled eight or even more times. The disadvantages of picking too large an unroll factor are:

1. All the loop instructions need not fit into the instruction cache leading to a lot of I-cache misses.
2. The higher the unroll factor, the higher the memory stride of memory instructions across iterations. If the memory stride exceeds cache line size, the effectiveness of prefetch decreases.
3. The resulting code is longer. Usually an upper bound must be chosen, an unroll count of 1000 is not likely to be a good idea since the compile time can go up significantly. Also excessive unrolling can adversely affect other optimizations which have bounds on the number of transformations they can make.

On the other hand if a small unroll factor is chosen the following problems can occur:

1. Much more time is spent executing the high latency branch instruction which closes the loop.
2. The short inner loop provides many fewer opportunities for optimization than longer inner loops. Where the inner loop has high latency instructions, the compiler can often have them execute in parallel with low latency time instructions. This may not be possible in very short loops.

One must keep in mind that the compiler is compiling loops that range from single instruction inner loops to loops that have scores or even hundreds of instructions, and so the compiler must compile code balance these considerations to achieve good unroll factors. To determine the unroll factor, the compiler considers the following in decreasing order of importance:

1. There is a maximum value of the unroll factor (which in the preferred embodiment of the invention is eight).
2. The number of instructions in the unrolled loop must not exceed a specific limit. This provides another upper bound to the unroll factor.
3. If there are references to previous indexed contents of the array such as was shown in FIGS. 10 through 12, an unroll factor suggested by this analysis (or a multiple of it) should be used.
4. If prefetch instructions are being generated (this is known based on a user defined flag), then try to pick an unroll factor that keeps the value of the strides of array references within the loop below the cache line size.
5. If the trip count is a constant known at compile time, then an unroll factor that eliminates the need for a compensation code loop should be selected. Typically this would be an unroll factor of 2, 4 or 8, although other numbers such as 3 or 5 might be possible.
6. If there is profile information, use that. If the profile informations says that the loop iterates on an average $\bar{k}$ times, if $\bar{k}$ is smaller than the maximum value of the unroll factor as dictated by the previous steps, use $\bar{k}$, else use the maximum value of the unroll factor.
7. If there are high latency operations within the loop such as divide and square root operations, use an unroll factor that will enhance the maximum overlap of these instructions. For instance, if the architecture has two divide units and the loop has a single divide instruction, the loop should be unrolled an even number of times so that both the divide units can be kept simultaneously busy.

The algorithm that computes the unroll factor tries to compute an optimal and acceptable unroll factor. The cost of a nonoptimal unroll factor is slower run time code. As discussed above, the algorithm is sensitive to profile data, number of instructions in the loop, architecture features like functional units and cache line size, interactions with other optimizations and constant trip counts.

Attention is directed to FIG. 13, which shows how the optimization algorithms presented here are implemented. At 201 the unroll factor is determined as described above. Next, at 203, a check is made for the special case where the trip count is known at compile time and is a multiple of the unroll factor. In this case, the unrolled loop code is generated from the original loop code, any middle exits are removed leaving only the final exit, and the unrolled code is output at 242. Because this is an unrolled loop which needs no compensation code, none is output. The other exit occurs at 203 where the trip count is not known at compile time, or the trip counts and unroll factor are such that compensation code must be generated. Control goes to 205 where the unrolled code is generated. For non-WHILE loops, the middle exits are removed leaving only the final exit. For a WHILE loop, the final exit is moved to the end of the loop (226) instead of the middle and all other exits removed. At this time a determination (at 229) is made using the unroll factor to determine if the compensation code should be output before or after the unrolled loop code. If it should be after, control goes to 244, otherwise control goes to 246. All of these three control paths then meet at 999, terminating the unrolling optimization.

FIG. 14 is a block schematic diagram of a compiler for a programmable machine in accordance with the invention. The compiler of FIG. 2*b* shows a loop unrolling module 30. The preferred embodiment of the invention provides a loop unrolling module that is placed within the compiler as shown in FIG. 2*b*. As shown in FIG. 14, the compiler comprises an analysis module 1301 for analyzing and unrolling loops within source applications. An optimizer module 1302 determines an optimum unroll factor in response to the analysis module. An unroll module 1303 generates an unrolled loop having said optimum unroll facto, while a compensation module 1304 generates and places any compensation code as required as a result of loop unroll optimization.

Although the invention is described herein with reference to the preferred embodiment, one skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention. Accordingly, the invention should only be limited by the claims included below.

We claim:

1. In a programmable machine, a compiler comprising:

an analysis module for analyzing and unrolling loops within source applications;

an optimizer module for determining an optimum unroll factor in response to said analysis module;

an unroll module for generating an unrolled loop having said optimum unroll factor; and

a compensation module for generating and placing any compensation code as required as a result of loop unroll optimization,

wherein said compensation module performs a placement calculation to determine whether to put said compensation code before the unrolled loop or after the unrolled loop.

2. The compiler of claim 1, wherein said compensation module ensures that said compensation code is executed at least once when said unrolled loop is executed.

3. The compiler of claim 1, wherein said unroll factor is responsive to a number of instructions in the loop.

4. The compiler of claim 1, wherein said unroll factor is responsive to resource usage within the loop.

5. The compiler of claim 1, wherein said unroll factor is responsive to prefetch distance of memory references within the loop.

6. The compiler of claim 1, wherein said unroll factor is responsive to profile information collected from previous executions of compiled code.

7. The compiler of claim 1, wherein said unroll factor is responsive to recurrence of memory references within the loop.

8. The compiler of claim 1, wherein said unroll factor is responsive to the number of instructions in the loop.

9. The compiler of claim 1, wherein a trip count for the loop is known at compile time, and wherein said optimizer module determines an unroll factor that executes the compensation code zero times and suppresses the generation of said compensation code.

10. The compiler of claim 1, wherein said placement calculation is responsive to the unroll factor computed for the loop.

11. The compiler of claim 1, wherein said loop is a loop having an early exit.

12. The compiler of claim 1, wherein loop unrolling is integrated with other low-level optimization phases.

13. The compiler of claim 12, wherein said other low-level optimization phases include any of prefetch instruction insertion, register reassociation, and instruction scheduling.

14. A method for unrolling loops, comprising the steps of:

determining a unroll factor;

generating an unrolled loop which always exits leaving a remaining trip count of at least one;

generating compensation code;

determining whether the compensation code should be placed before the unrolled loop or after the unrolled loop; and

determining if the trip count is a power of two; and if it is placing the compensation code before the unrolled loop.

15. The method of claim 14, wherein the loop to be unrolled is a loop having an early exit.

16. The method of claim 15, wherein the loop having an early exit after unrolling is transformed into a loop having an exit at its end, and from which all intermediate exits have been removed.

* * * * *

US005797013A

United States Patent [19]

Mahadevan et al.

[11] Patent Number: 5,797,013

[45] Date of Patent: Aug. 18, 1998

[54] INTELLIGENT LOOP UNROLLING

[75] Inventors: Uma Mahadevan, Sunnyvale; Lacky Shah, Fremont, both of Calif.

[73] Assignee: Hewlett-Packard Company, Palo Alto, Calif.

[21] Appl. No.: 564,514

[22] Filed: Nov. 29, 1995

[51] Int. Cl.^{6} G06F 9/45

[52] U.S. Cl. 395/709; 395/588

[58] Field of Search 395/705, 709, 395/588, 580

[56] References Cited

U.S. PATENT DOCUMENTS

5,265,253	11/1993	Yamada	395/700
5,367,651	11/1994	Smith et al.	395/700
5,386,562	1/1995	Jain et al.	395/650

OTHER PUBLICATIONS

"A Comparative Evaluation of Software Techniques to Hide Memory Latency", John et al., Proc. of the 28th Ann. Hawaii Int'l Conf., 1995, pp. 229–238.

"Schedule driven Loop Unrolling for Parallel Processors", System Sciences, 1991 Annual Hawaii Int'l Conference, 1991, vol. II pp. 458–467.

"Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler", Davidson et al., Dept of Comp. Science, Univ. of Va. pp. 1–14.

"Unrolling Loops in Fortran," Dongarra et al., Soft. Practice and Experience, vol. 9, 1979, pp. 219–226.

Hendren et al., "Designing Programming Languages for the Analyzability of Pointer Data Structures," Comput. Lang., vol. 19, No. 2, pp. 119–134 (1993).

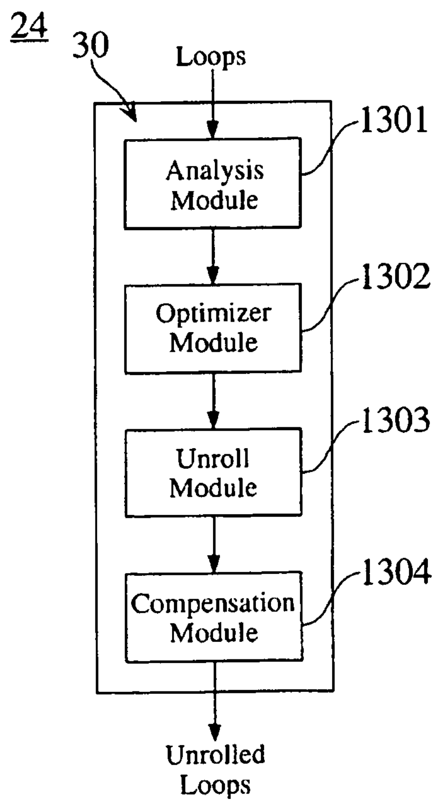
Weiss et al., "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," ACM, pp. 105–109 (1987).

Primary Examiner—Emanuel Todd Voeltz
Assistant Examiner—Kakali Chaki

[57] ABSTRACT

A compiler facilitates efficient unrolling of loops and enables the elimination of extra branches from the loops, including the elimination of conditional branches from unrolled loops with early exits. Unrolling also enhances other optimizations, such as prefetch, scalar replacement, and instruction scheduling. The unroll factor is calculated to determine the amount of loop expansion and the optimum location to place compensation code to complete the original loop count, i.e. before or after the unrolled loop. The compiler is applicable, for example, to modern RISC architectures, where the latency of memory references and branches is higher than that of integer and floating point arithmetic instructions.

16 Claims, 13 Drawing Sheets



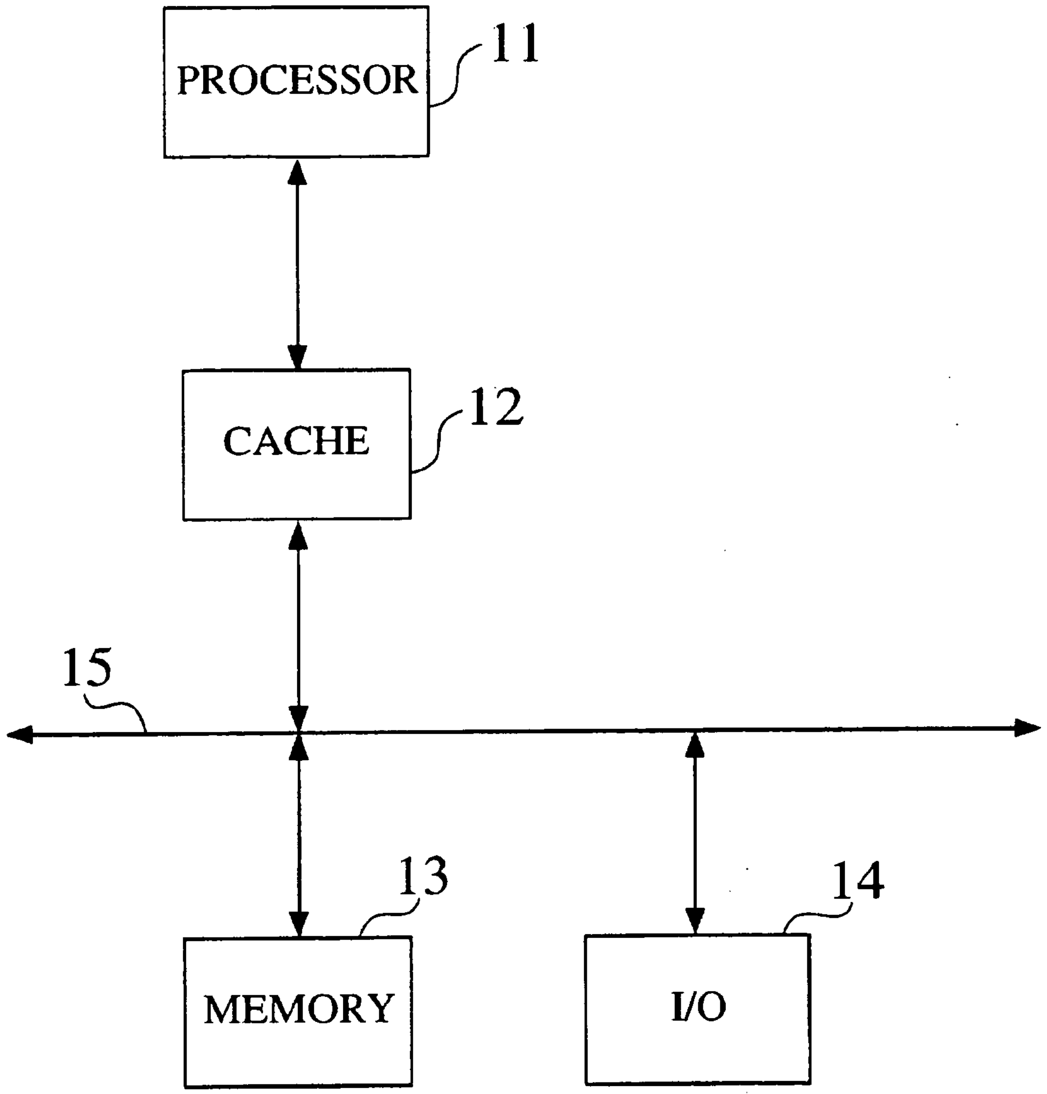


Fig. 1 (PRIOR ART)

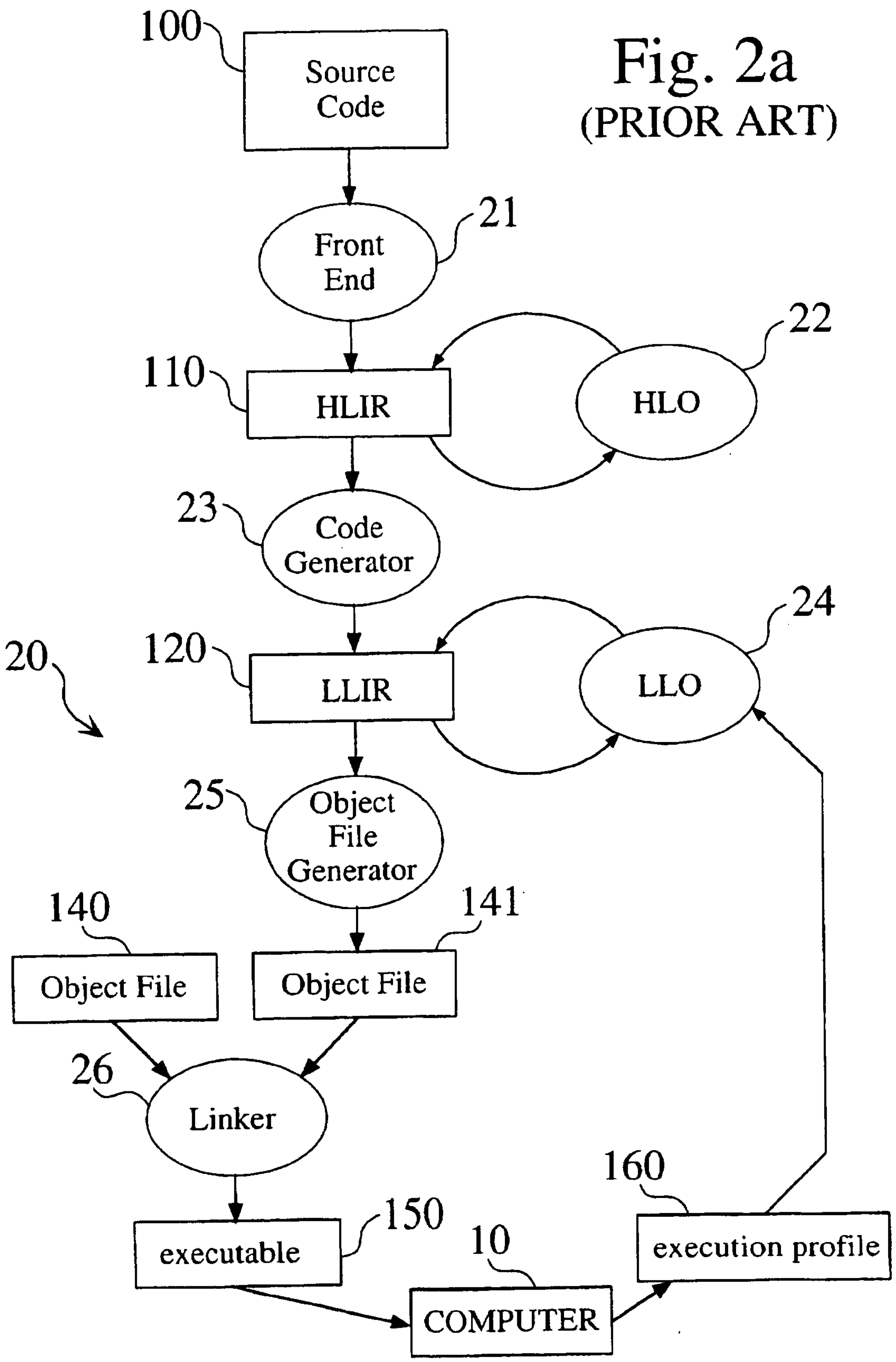


Fig. 2a
(PRIOR ART)

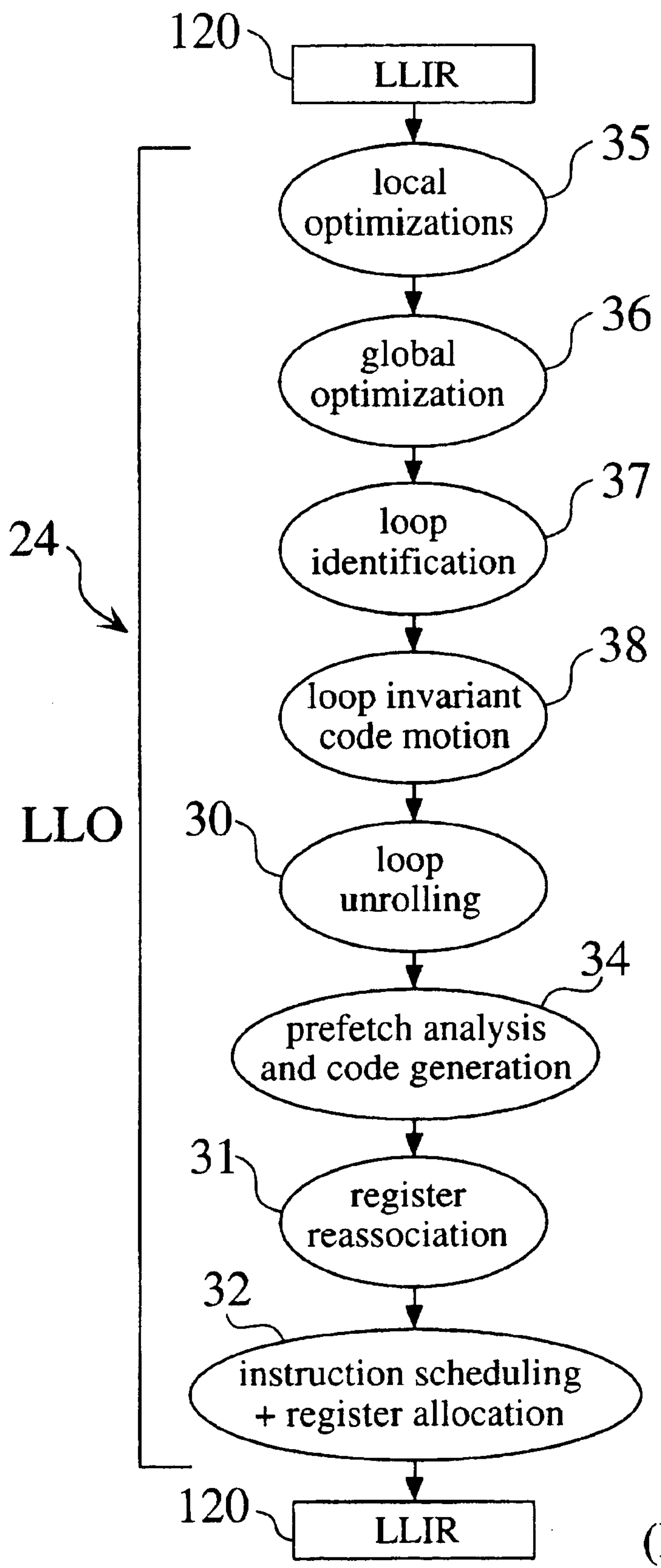


Fig. 2b
(PRIOR ART)

```
    I=N                                          150
BEGIN_LOOP:
 A[I] = B[I] + X
 I = I -1
 IF  (I>0)  GOTO BEGIN_LOOP;
END_LOOP:
PRINT I
```

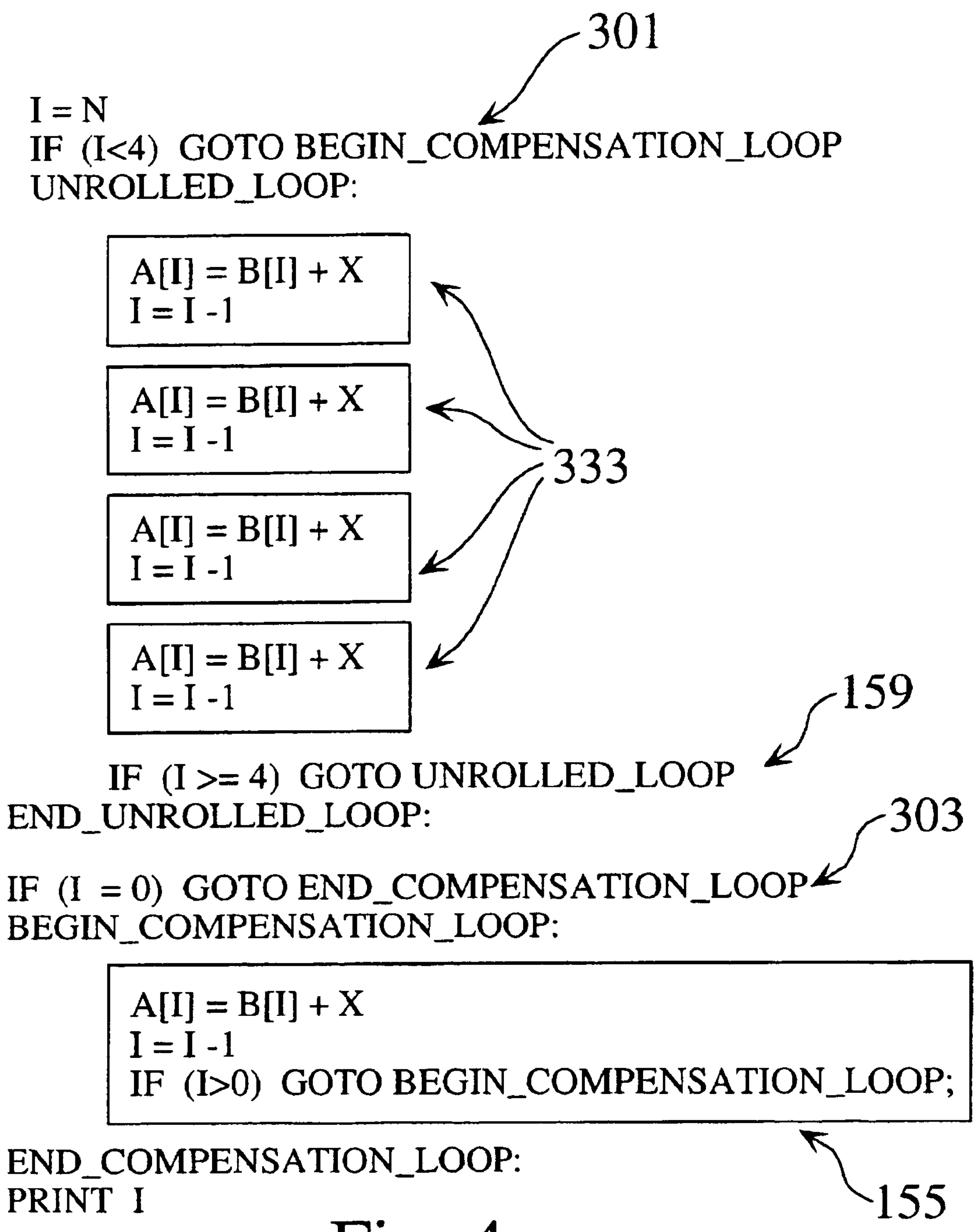
Fig. 3 (PRIOR ART)

```
I = N
BEGIN_LOOP:
    A[I] = B[I] + K                401
    I = I -1
    IF ( I < 0) GOTO END_LOOP      403
       C[I] = A[I] + X             405
       GOTO BEGIN_LOOP
  END_LOOP:
  PRINT I
```

Fig. 7a (PRIOR ART)

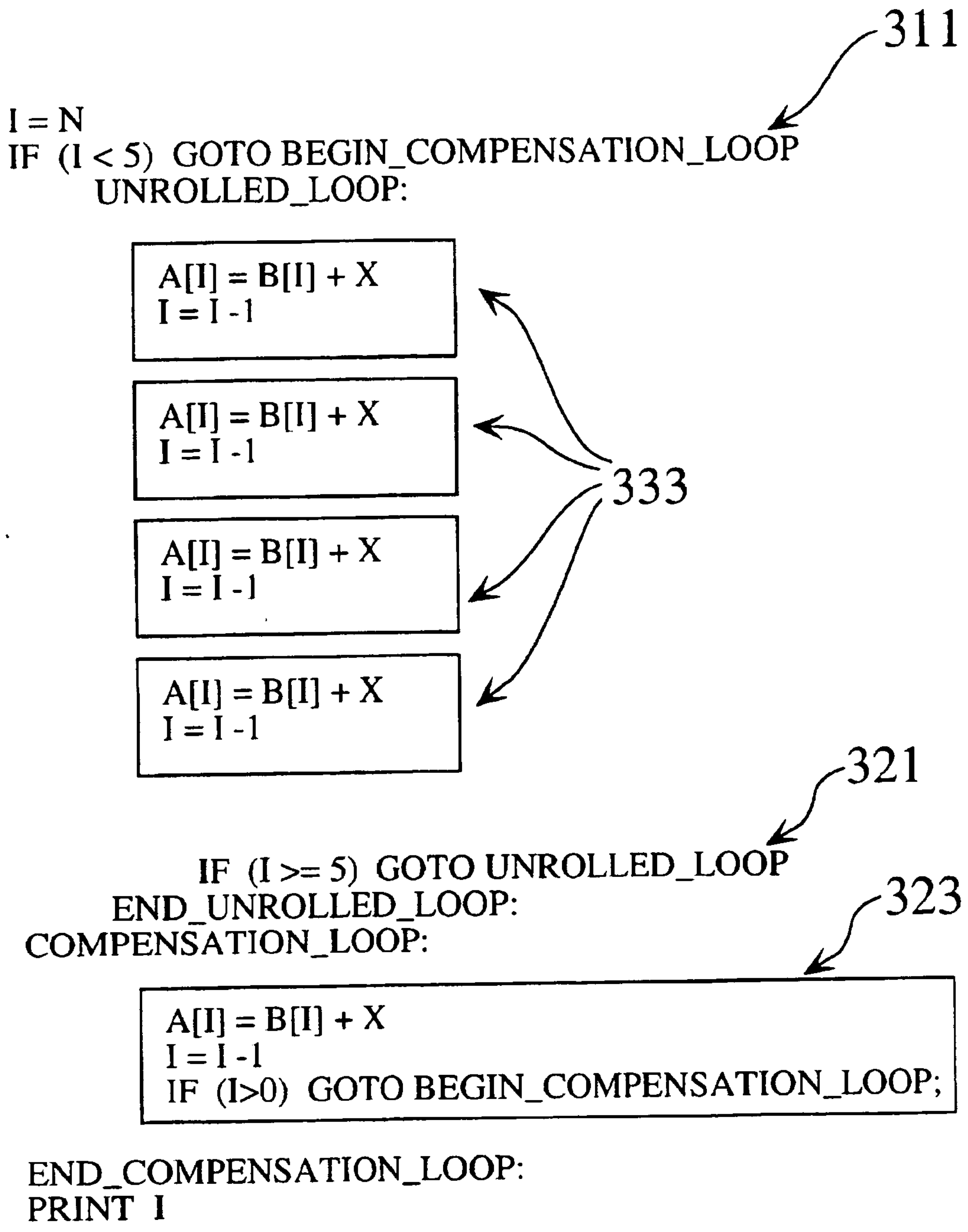
```
BEGIN_LOOP:
     A[I] = A[I-2] / B[I]          367
     I = I  + 1;
   IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 9 (PRIOR ART)



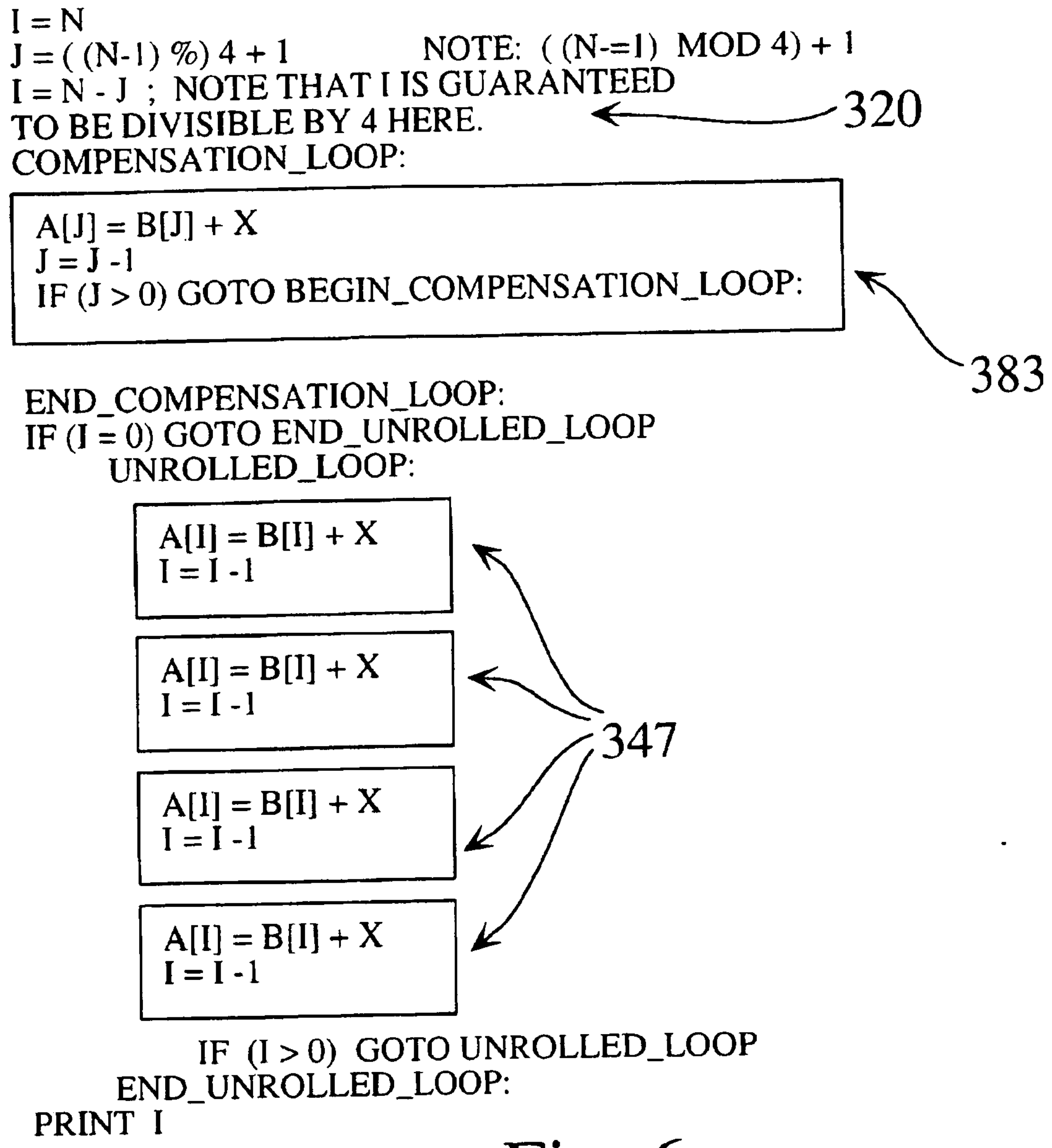
```
I = N
IF (I<4) GOTO BEGIN_COMPENSATION_LOOP
UNROLLED_LOOP:
    A[I] = B[I] + X
    I = I -1
    A[I] = B[I] + X
    I = I -1
    A[I] = B[I] + X
    I = I -1
    A[I] = B[I] + X
    I = I -1
    IF (I >= 4) GOTO UNROLLED_LOOP
END_UNROLLED_LOOP:

IF (I = 0) GOTO END_COMPENSATION_LOOP
BEGIN_COMPENSATION_LOOP:
    A[I] = B[I] + X
    I = I -1
    IF (I>0) GOTO BEGIN_COMPENSATION_LOOP;
END_COMPENSATION_LOOP:
PRINT I
```

Fig. 4
(PRIOR ART)



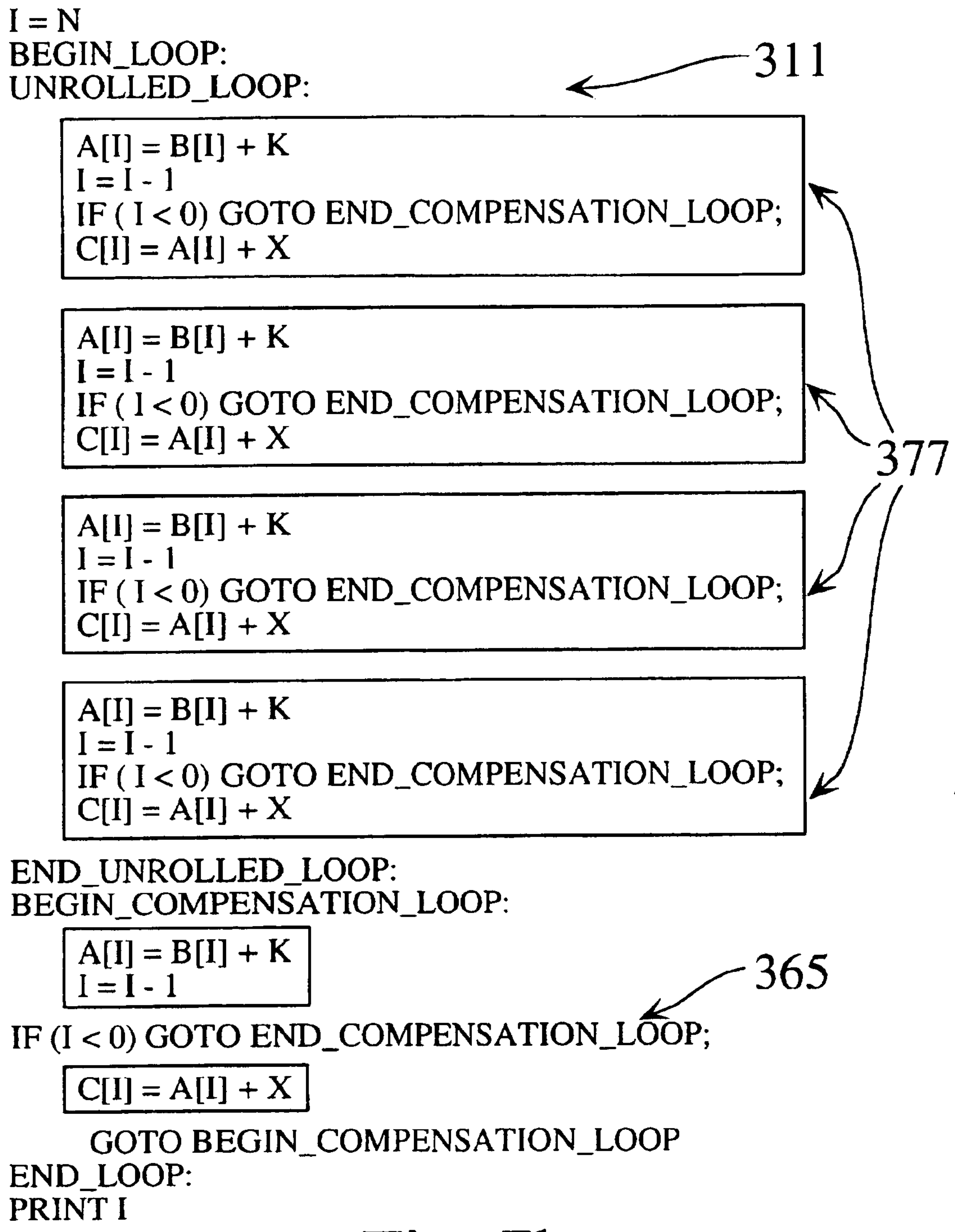
```
I = N
IF  (I < 5)  GOTO BEGIN_COMPENSATION_LOOP
      UNROLLED_LOOP:

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

          A[I] = B[I] + X
          I = I -1

             IF  (I >= 5)  GOTO UNROLLED_LOOP
       END_UNROLLED_LOOP:
 COMPENSATION_LOOP:

          A[I] = B[I] + X
          I = I -1
          IF  (I>0)  GOTO BEGIN_COMPENSATION_LOOP;

 END_COMPENSATION_LOOP:
 PRINT  I
```

Fig. 5



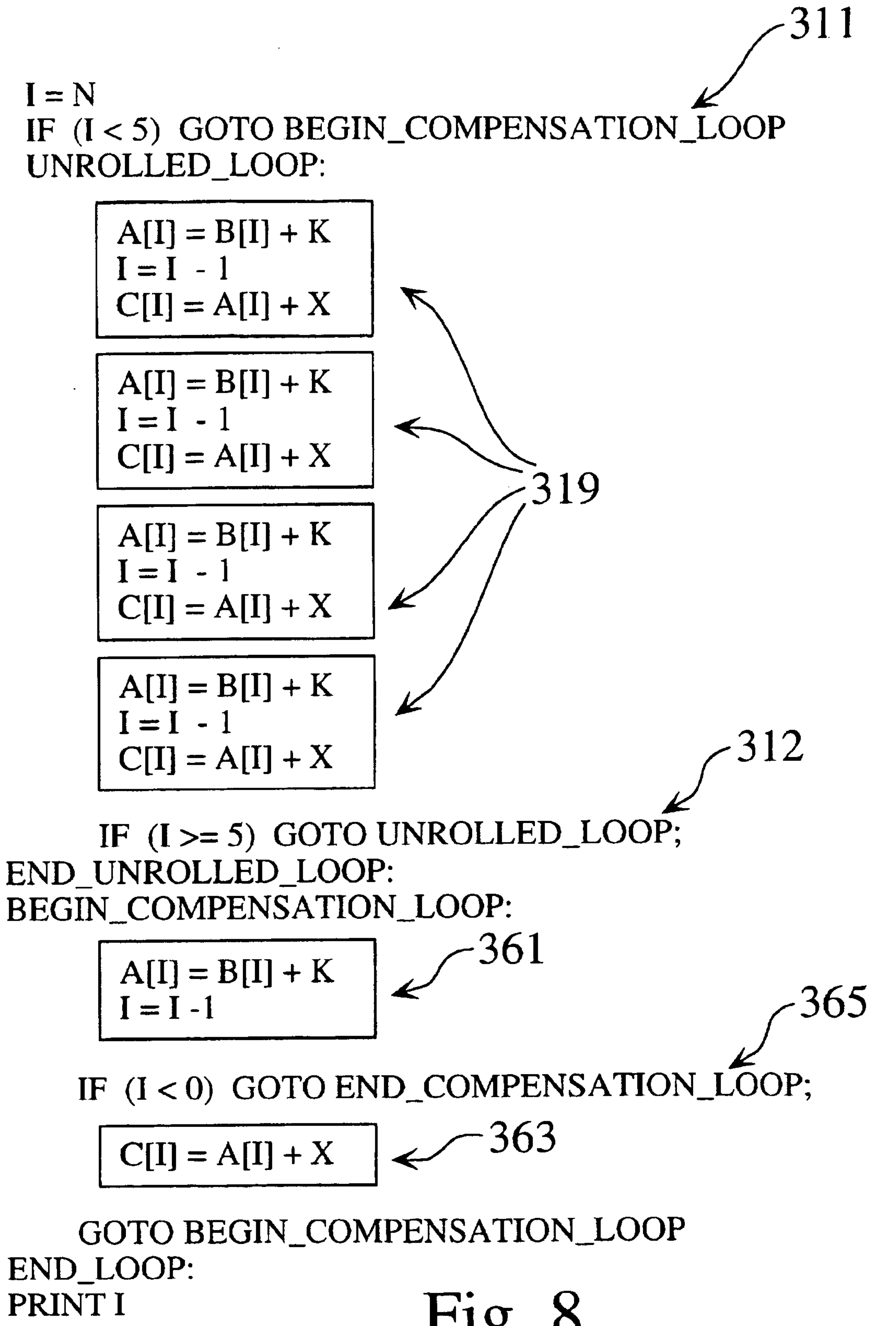
```
I = N
J = ( (N-1) %) 4 + 1          NOTE: ( (N-=1) MOD 4) + 1
I = N - J ; NOTE THAT I IS GUARANTEED
TO BE DIVISIBLE BY 4 HERE.
COMPENSATION_LOOP:
  A[J] = B[J] + X
  J = J -1
  IF (J > 0) GOTO BEGIN_COMPENSATION_LOOP:

END_COMPENSATION_LOOP:
IF (I = 0) GOTO END_UNROLLED_LOOP
     UNROLLED_LOOP:
        A[I] = B[I] + X
        I = I -1
        A[I] = B[I] + X
        I = I -1
        A[I] = B[I] + X
        I = I -1
        A[I] = B[I] + X
        I = I -1
          IF (I > 0) GOTO UNROLLED_LOOP
     END_UNROLLED_LOOP:
 PRINT I
```

Fig. 6



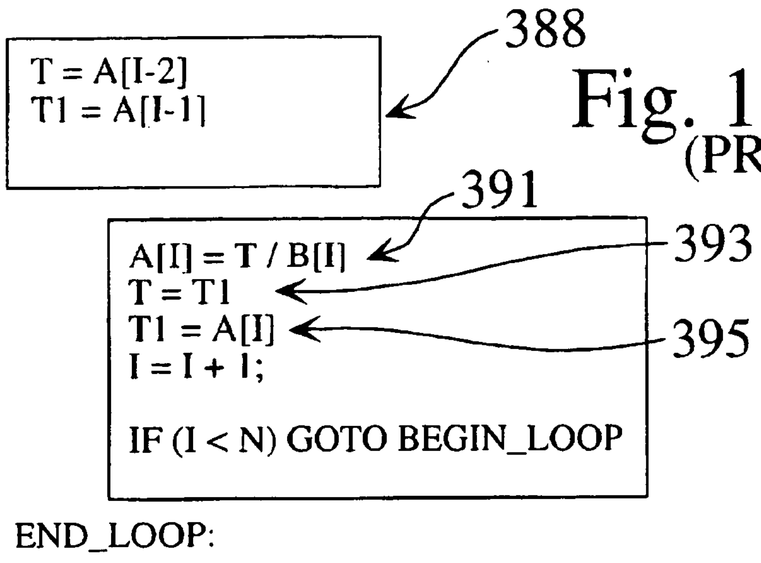
```
I = N
BEGIN_LOOP:
UNROLLED_LOOP:
    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X

    A[I] = B[I] + K
    I = I - 1
    IF ( I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X
END_UNROLLED_LOOP:
BEGIN_COMPENSATION_LOOP:
    A[I] = B[I] + K
    I = I - 1
IF (I < 0) GOTO END_COMPENSATION_LOOP;
    C[I] = A[I] + X
      GOTO BEGIN_COMPENSATION_LOOP
END_LOOP:
PRINT I
```

Fig. 7b (PRIOR ART)



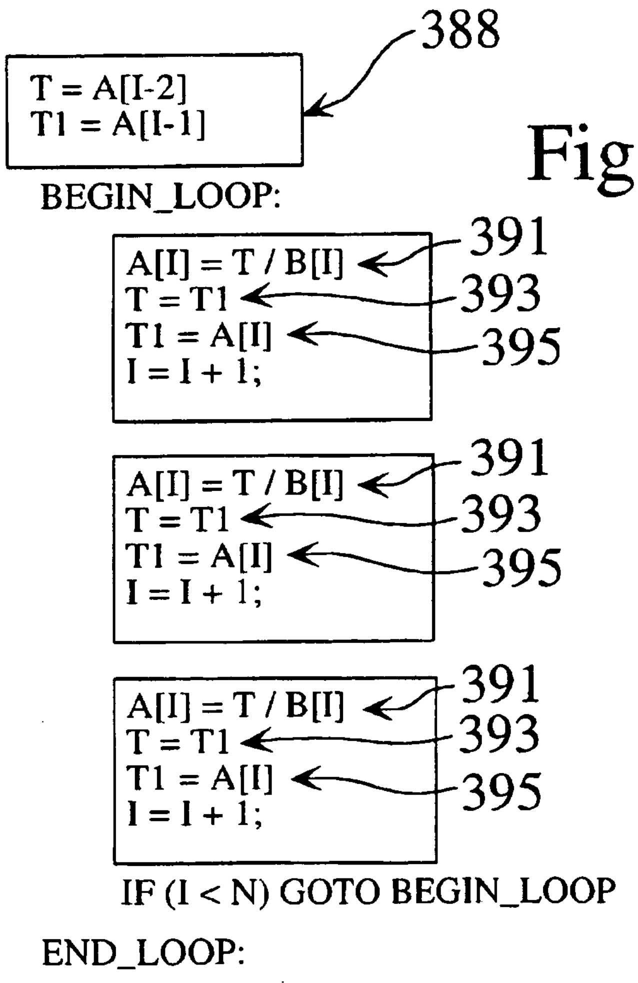
```
I = N
IF  (I < 5)  GOTO BEGIN_COMPENSATION_LOOP
UNROLLED_LOOP:
        A[I] = B[I] + K
        I = I  - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I  - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I  - 1
        C[I] = A[I] + X

        A[I] = B[I] + K
        I = I  - 1
        C[I] = A[I] + X

        IF  (I >= 5)  GOTO UNROLLED_LOOP;
END_UNROLLED_LOOP:
BEGIN_COMPENSATION_LOOP:
        A[I] = B[I] + K
        I = I -1

      IF  (I < 0)  GOTO END_COMPENSATION_LOOP;

        C[I] = A[I] + X

      GOTO BEGIN_COMPENSATION_LOOP
END_LOOP:
PRINT I
```

Fig. 8



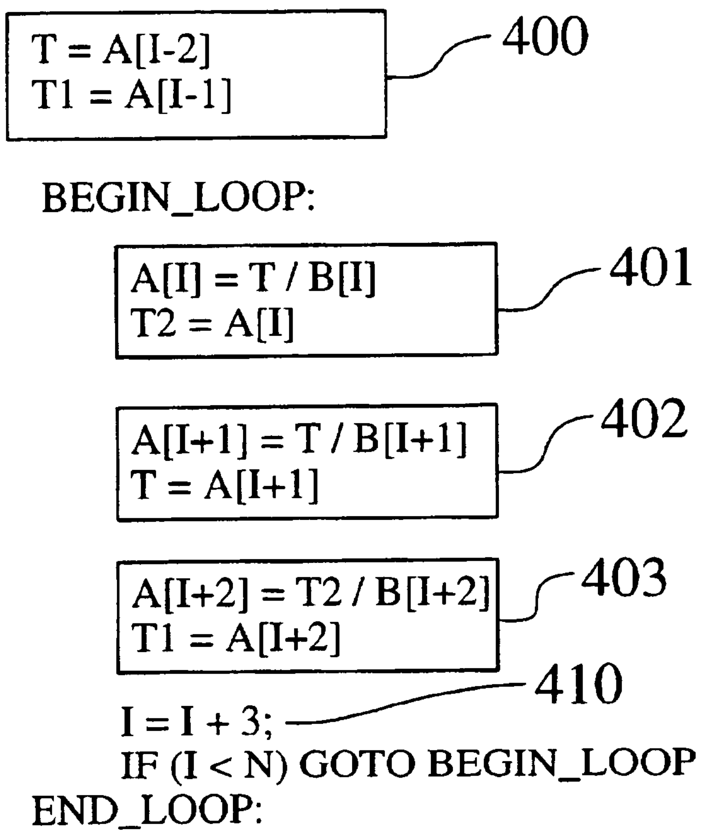
```
T = A[I-2]
T1 = A[I-1]

A[I] = T / B[I]
T = T1
T1 = A[I]
I = I + 1;

IF (I < N) GOTO BEGIN_LOOP

END_LOOP:
```

Fig. 10 (PRIOR ART)



```
T = A[I-2]
T1 = A[I-1]
BEGIN_LOOP:
    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;

    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;

    A[I] = T / B[I]
    T = T1
    T1 = A[I]
    I = I + 1;
    IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 11 (PRIOR ART)



```
T = A[I-2]
T1 = A[I-1]

BEGIN_LOOP:
      A[I] = T / B[I]
      T2 = A[I]

      A[I+1] = T / B[I+1]
      T = A[I+1]

      A[I+2] = T2 / B[I+2]
      T1 = A[I+2]

      I = I + 3;
      IF (I < N) GOTO BEGIN_LOOP
END_LOOP:
```

Fig. 12

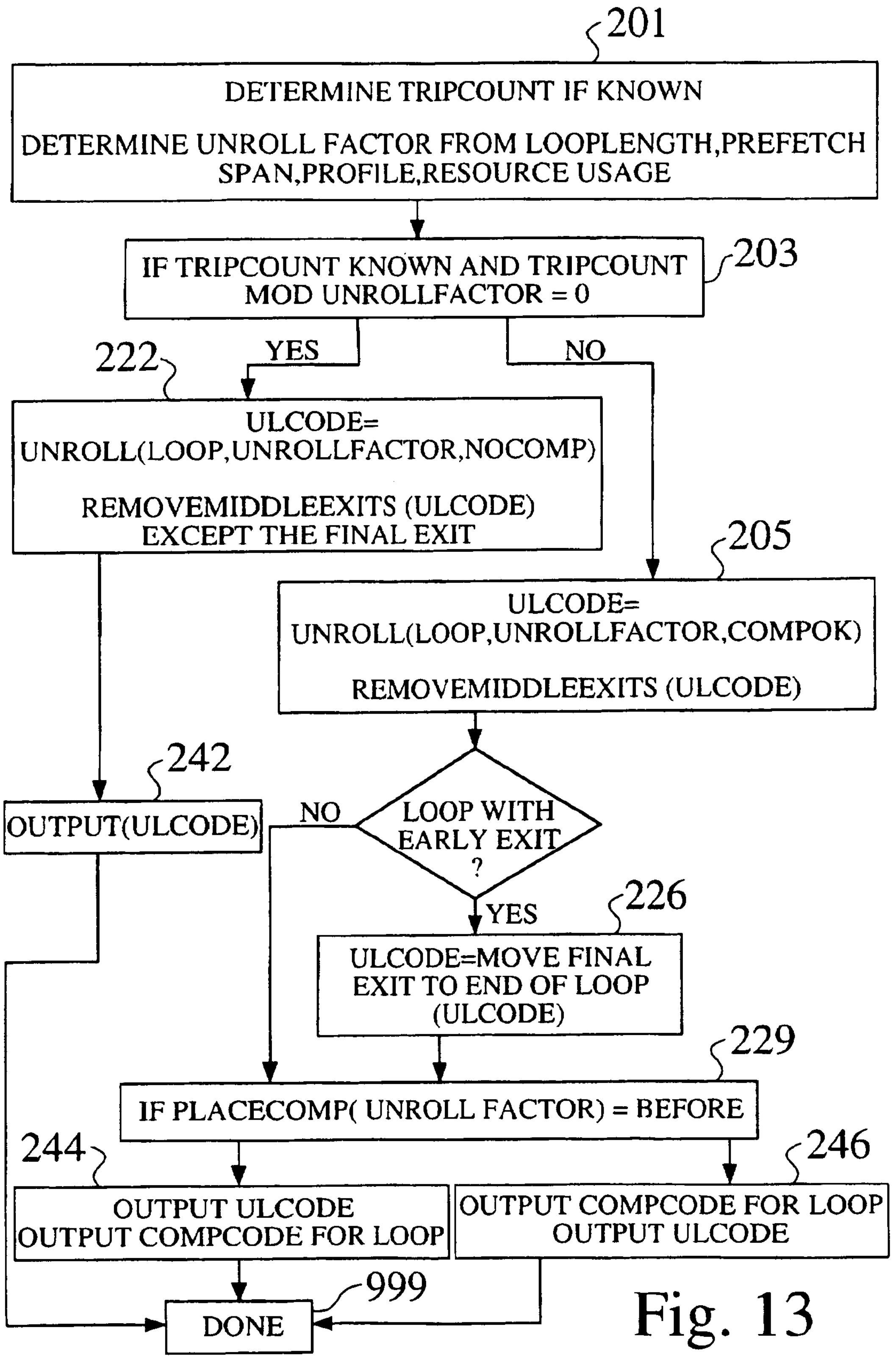

201
DETERMINE TRIPCOUNT IF KNOWN
DETERMINE UNROLL FACTOR FROM LOOPLENGTH,PREFETCH SPAN,PROFILE,RESOURCE USAGE
203
IF TRIPCOUNT KNOWN AND TRIPCOUNT MOD UNROLLFACTOR = 0
YES
NO
222
ULCODE= UNROLL(LOOP,UNROLLFACTOR,NOCOMP)
REMOVEMIDDLEEXITS (ULCODE) EXCEPT THE FINAL EXIT
205
ULCODE= UNROLL(LOOP,UNROLLFACTOR,COMPOK)
REMOVEMIDDLEEXITS (ULCODE)
242
OUTPUT(ULCODE)
NO
LOOP WITH EARLY EXIT ?
YES
226
ULCODE=MOVE FINAL EXIT TO END OF LOOP (ULCODE)
229
IF PLACECOMP(UNROLL FACTOR) = BEFORE
244
OUTPUT ULCODE OUTPUT COMPCODE FOR LOOP
246
OUTPUT COMPCODE FOR LOOP OUTPUT ULCODE
999
DONE

Fig. 13

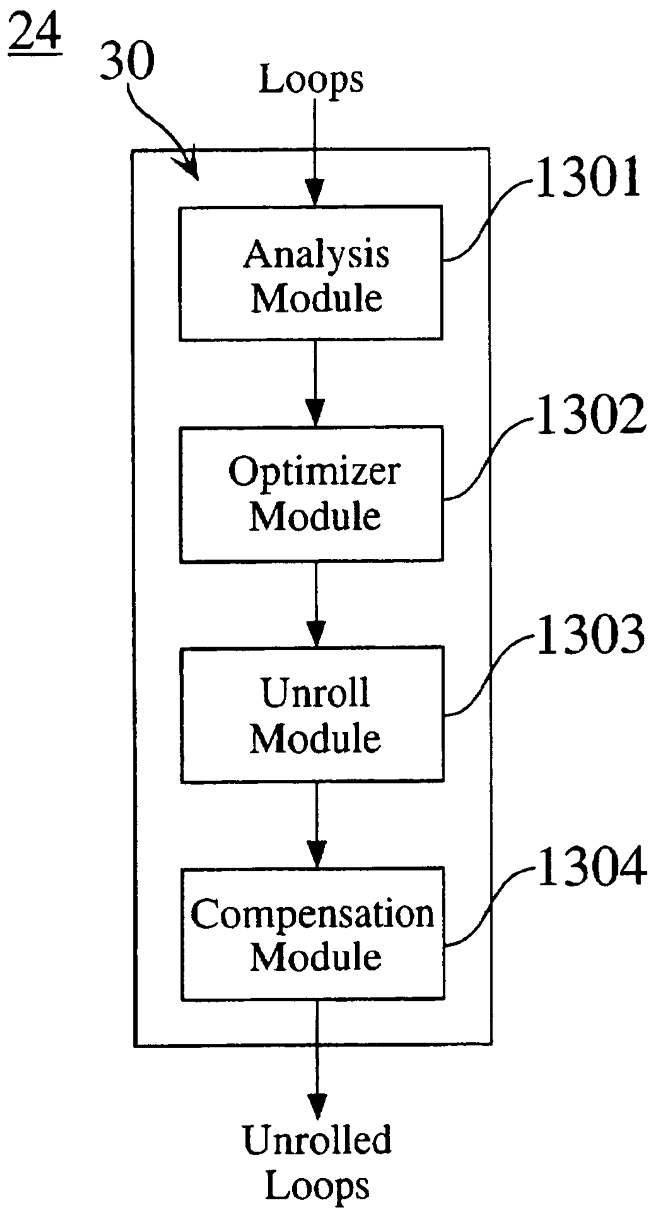

24
30
Loops
Analysis Module
1301
Optimizer Module
1302
Unroll Module
1303
Compensation Module
1304
Unrolled Loops

Fig. 14

INTELLIGENT LOOP UNROLLING

BACKGROUND OF THE INVENTION

1. Technical Field

The invention relates to compilers. More particularly, the invention relates to techniques for unrolling computation loops in a compiler so as to generate code which executes faster.

2. Description of The Prior Art

FIG. 1 is a block schematic diagram of a uniprocessor computer architecture, including a processor cache. In the figure, a processor **11** includes a cache **12** which is in communication with a system bus **15**. A system memory **13** and one or more I/O devices **14** are also in communication with the system bus.

FIG. 2*a* is a block schematic diagram of a software compiler **20**, for example as may be used in connection with the computer architecture shown in FIG. 1. The compiler Front End component **21** reads a source code file (**100**) and translates it into a high level intermediate representation (**110**). A high level optimizer **22** optimizes the high level intermediate representation **110** into a more efficient form. A code generator **23** translates the optimized high level intermediate representation to a low level intermediate representation (**120**). The low-level optimizer **24** converts the low level intermediate representation (**120**) into a more efficient (machine-executable) form. Finally, an object file generator **25** writes out the optimized low-level intermediate representation into an object file (**141**). The object file (**141**) is processed along with other object files (**140**) by a linker **26** to produce an executable file (**150**), which can be run on the computer **10**. In the invention described herein, it is assumed that the executable file (**150**) can be instrumented by the compiler (**20**) and linker (**26**) so that when it is run on the computer **10**, an execution profile (**160**) may be generated, which can then be used by the low level optimizer **24** to better optimize the low-level intermediate representation (**120**). The compiler **20** is discussed in greater detail below.

The compiler is the piece of software that translates source code, such as C, BASIC, or FORTRAN, into a binary image that actually runs on a machine. Typically the compiler consists of multiple distinct phases, as discussed above in connection with FIG. 2*a*. One phase is referred to as the front end, and is responsible for checking the syntactic correctness of the source code. If the compiler is a C compiler, it is necessary to make sure that the code is legal C code. There is also a code generation phase, and the interface between the front-end and the code generator is a high level intermediate representation. The high level intermediate representation is a more refined series of instructions that need to be carried out. For instance, a loop might be coded at the source level as:

```
for(I=0,I<10,I=I+1),
```

which might in fact be broken down into a series of steps, e.g. each time through the loop, first load up I and check it against 10 to decide whether to execute the next iteration.

A code generator takes this high level intermediate representation and transforms it into a low level intermediate representation. This is closer to the actual instructions that the computer understands. An optimizer component of a compiler must preserve the program semantics (i.e. the meaning of the instructions that are translated from source code to an high level intermediate representation, and thence to a low level intermediate representation and ultimately an executable file), but rewrites or transforms the code in a way that allows the computer to execute an equivalent set of instructions in less time.

Modern compilers are structured with a high level optimizer (HLO) that typically operates on a high level intermediate representation and substitutes in its place a more efficient high level intermediate representation of a particular program that is typically shorter. For example, an HLO might eliminate redundant computations. With the low level optimizer (LLO), the objectives are the same as the HLO, except that the LLO operates on a representation of the program that is much closer to what the machine actually understands.

FIG. 2*b* is a block diagram showing a low level optimizer for a compiler, including a loop unrolling component **30** according to the invention. The low level optimizer **24** may include any combination of known optimization techniques, such as those that provide for local optimization **35**, global optimization **36**, loop identification **37**, loop invariant code motion **38**, prefetch **34**,register reassociation **31**, and instruction scheduling **32**.

Source programs translated into machine code by compilers consists of loops, e.g. DO loops, FOR loops, and WHILE loops. Optimizing the compilation of such loops can have a major effect on the run time performance of the program generated by the compiler. In some cases, a significant amount of time is spent doing such bookkeeping functions as loop iteration and branching, as opposed to the computations that are performed within the loop itself. These loops often implement scientific applications that manipulate large arrays and data instructions, and run on high speed processors.

This is particularly true on modern processors, such as RISC architecture machines. The design of these processors is such that in general the arithmetic operations operate a lot faster than memory fetch operations. This mismatch between processor and memory speed is a very significant factor in limiting the performance of microprocessors. Also, branch instructions, both conditional and unconditional, have an increasing effect on the performance of programs. This is because most modern architectures are super-pipelined and have some sort of a branch prediction algorithm implemented. The aggressive pipelining makes the branch misprediction penalty very high. Arithmetic instructions are interregister instructions that can execute quickly, while the branch instructions, because of mispredictions, and memory instructions such as loads and stores, because of slower memory speeds, can take a longer time to execute.

Modern compilers perform code optimization. Code optimization consists of several operations that improve the speed and size of the complied code, while maintaining semantic equivalence. Common optimizations include:

- prefetching data so that they are available in cache memory when needed;
- detecting calculations as computing constants and performing the calculation at compile time;
- scalar replacement which keeps the value of a variable in a register within the loop;
- moving calculations outside of loops where possible; and
- performing code scheduling, which consists of rearranging the order of and modifying instructions to achieve faster running but semantically equivalent code.

Many modern compilers also employ an optimizing technique known as loop unrolling to generate faster running code. In its essence, loop unrolling takes the inner loop, i.e. the code between the beginning and the end of the loop, and repeats it in the inner loop some number of times, e.g. four times. It then executes the unrolled loop one-fourth as many times as it would have executed the original loop. The number of times the loop is replicated within the unrolled loop is called the unroll factor. Because the number of times the original loop is executed is not always divisible by the unroll factor, a compensation loop code often has to be generated to execute the remaining of instructions of the original loop that are not executed by the unrolled loop.

As discussed above, such loops as DO, FOR and WHILE loops are common in programs, especially in scientific and other time-consuming programs. Frequently 80% of the running time of a program can be in a few small loops. As a result anything that can speed up such loops is of great value in making a more efficient compiler.

Consider the simple loop shown on FIG. 3. The three instructions in the inner loop **150** are executed N times. According to the prior art, this loop can be unrolled with an unroll factor of four to produce the code shown on FIG. 4, where the inner loop (less the exit condition) is replicated 4 times 333. This loop is also followed by a test **303** to see if the full original loop has been completed, and a compensation loop **155** which is executed to the complete the original loop trip count if it has not been completed.

An inspection of the loop shows that it is semantically equivalent to the loop of FIG. 3 because the same function is performed and the same result is achieved. However, the loop of FIG. 4 runs much faster for several reasons. First, the conditional branch **159** which exits the loop is executed only once for each time through the unrolled loop rather than once for each time through the original loop. Assuming an unroll factor of four as is shown here, this saves ¾ of a conditional branch per original loop iteration. More importantly, other compiler optimizations interact with loop unrolling and are able to do a much better job of optimizing the unrolled loop, such as that identified by numeric designator **333**, as compared to the original loop, identified by numeric designator **150**. In an unrolled loop, there are more operations that could be scheduled in parallel, more opportunity to do scalar replacement and other optimizations, and more possibilities to do prefetching.

The tests identified by numeric designators **301** and **303** are also of interest. These are the conditional branches which have a higher probability of being mispredicted. Anything that can be done to eliminate one of them will be very useful.

Prior art loop unrolling techniques have certain disadvantages. For example, J. J. Dongara and A. R. Hinds, *Unrolling Loops in FORTRAN*, describes how one can unroll loops manually by duplicating code. This is an early solution to optimizing code that it is not even implemented by the compiler.

S. Weiss and J. E. Smith, *A Study of scalar compilation techniques for pipeline supercomputers*, discuss unrolling in the compiler. Here the authors address the simple situations:

a) Cases where the loop count is known at compile time. They do not address loop unrolling when the loop count is only known at run time.

b) Cases where the loop exit appears only at the beginning or end of the loop. They do not address the situation of unrolling loops with early exits (loops whose exit may occur in the middle of the loop).

The authors do not address the following issues:

a) Determining whether to place the compensation code before or after the unrolled loop.

b) Tuning of the iteration count to reduce branch misprediction.

c) Factors that affect the unroll factor.

L. J. Hendren and G. R. Gao, *Designing Programming Languages for the Analyzability of Pointer Data Structures*, addresses the issue of unrolling loops as part of compiler optimization. They do not however discuss:

a) Unrolling loops with early exits;

b) Tuning of the iteration count to reduce branch misprediction;

c) Factors that affect the unroll factor;

d) Whether to place a compensation code at the beginning or end of the loops; and

e) Compiling loops whose trip count is only known at run time.

J. Davidson, S. Jinturkar, *Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler*, Dept. of Computer Science, Thornton Hall, University of Virginia disclose a code transformation, referred to as aggressive loop unrolling, in a retargetable optimizing compiler where the loop bounds are not known at compile time. Various factors were analyzed to determine how and when loop unrolling should be applied, resulting in an algorithm for loop unrolling in which execution-time counting loops (i.e. a counting loop whose iteration count is not trivially known at compile time) are unrolled and loops having complex control-flow are unrolled. However, they do not discuss:

a) Unrolling loops having early exits;

b) Tuning of the iteration count to reduce branch misprediction

c) Factors that affect the unroll factor; and

d) Whether to place a compensation code at the beginning or end of the loops.

Another part of the loop unrolling prior art is shown on FIG. 7, which illustrates a loop having an early exit (also referred to as a WHILE loop), consisting of an exit test and branch **403** in the middle of the loop between computations **401** and **405**. According to the prior art, the code, including the exit **403**, is replicated four times in an unrolled loop. The number of branches in the loop are however not reduced with this optimization.

While some of the techniques discussed in the prior art are applicable to compilers for all computers, only some of them are particularly applicable for modern RISC computers, where branch instructions form a lot bigger bottleneck than in earlier technologies. Also compilers for RISC architectures are a lot more aggressive and the interactions of various optimizations plays a key role in the quality of the final code.

SUMMARY OF THE INVENTION

The invention provides a new compiler that can unroll more loops than previous algorithms. It also significantly reduces the number of branch instructions by cleverly handling the iteration count and by converting loops with early exits to regular FOR loops. The invention also provides for computing the unroll factor and the placement of the compensation loop by taking a lot of other optimizations into consideration.

The compiler:

Eliminates time consuming conditional branch instructions from the compensation code loop by replacing the conditional exit of the main unrolled loop to always exit with at least one iteration which has yet to be executed by the compensation code. This eliminates the need to test for zero remaining loops.

Determines whether it is better to place the compensation code at the beginning or the end of the unrolled loop according to which one would likely provide the better optimization. Generally, it prefers to put the compensation loop in front of the main loop if the unroll factor is a power of two and after the main loop if the unroll factor is not a power of two.

Computes the unroll factor by taking into account the interactions of other optimizations like prefetch, scalar replacement and register allocation, and also taking into account hardware features like number of functional units. Unrolling loops over-aggressively or under-aggressively can inhibit other optimizations or make them less effective.

Converts loops with early exit to loops with exit at the end to apply more efficient optimizations to the loop. It does this by ensuring that the compensation code is always executed at least once, enabling the compiler to eliminate the exit tests from the unrolled loop.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a block schematic diagram of a uniprocessor computer architecture including a processor cache;

FIG. 2*a* shows a block schematic diagram of a modern software compiler;

FIG. 2*b* shows schematic diagram of the low level optimizer;

FIG. 3 shows a simple program loop;

FIG. 4 shows the loop of FIG. 3 that has been unrolled four times according to the prior art;

FIG. 5 shows the loop of FIG. 3 unrolled according to the invention and having an eliminated branch;

FIG. 6 shows an unrolled loop having pre-loop compensation code;

FIG. 7*a* shows a simple WHILE loop;

FIG. 7*b* shows the WHILE loop of FIG. 7*a* that has been unrolled according to the prior art;

FIG. 8 shows the WHILE loop of FIG. 7*a* that has been unrolled according to the invention;

FIG. 9 shows a schematic representation of a simple loop;

FIG. 10 shows the loop of FIG. 9 with scalar replacement according to the prior art;

FIG. 11 shows the loop of FIG. 10 unrolled;

FIG. 12 shows a schematic representation of the loop of FIG. 11 after copy elimination;

FIG. 13 shows a schematic representation of the compiler logic which determines the unroll factor, compensation code placement, and other optimizations; and

FIG. 14 is a block schematic diagram of a compiler for a programmable machine in accordance with the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention provides a new compiler that features smart unrolling of loops.

The invention provides a prefetch driver 34 that operates in concert with such known techniques. The following discussion pertains to the various elements of the low level optimizer shown on FIG. 2*b:*

Local optimizations include code improving transformations that are applied on a basic block by basic block basis. For purposes of the discussion herein, a basic block corresponds to the longest contiguous sequence of machine instructions without any incoming or outgoing control transfers, excluding function calls. Examples of local optimizations include local common sub-expression elimination (CSE), local redundant load elimination, and peephole optimization.

Global optimizations include code improving transformations that are applied based on analysis that spans across basic block boundaries. Examples include global common sub-expression elimination, loop invariant code motion, dead code elimination, register allocation and instruction scheduling.

Loop invariant code motion is the identification of instructions located with a loop that compute the same result on every loop iteration and the re-positioning of such instructions outside the loop body.

Register allocation and instruction scheduling is the process of assigning hardware registers to symbolic instruction operands and the re-ordering of instructions to minimize run-time pipeline stalls where the processor must wait on a memory fetch from main memory or wait for the completion of certain complicated instructions that take multiple cycles to execute (eg. divide, square root instructions).

One important phase of the compiler identifies loops and access patterns to estimate how many cycles are devoted to loop iterations. In the invention, the compiler translates the higher level application code into an instruction stream that the processor executes, and in the process of this translation the compiler unrolls loops.

The longer unrolled loops allow the compiler to provide several advantages, such as:

1) It eliminates the extra branch exits. This saves CPU cycles by not having to execute the branch instructions and also helps reduce branch misprediction. Why this is important is evident if one considers most modern RISC architectures. These architectures have a long pipeline that is fed by an instruction fetch mechanism. When the fetch mechanism encounters a branch, it tries to predict if the branch is going to be taken or not. It then fetches instructions based on this prediction. The prediction is necessary to keep the pipeline from stalling. If the architecture's prediction is correct (this is determined when the branch instruction completes execution which is a few cycles after it has been fetched), then everything works fine; else all the instructions that have been fetched after the branch are discarded and the new instructions fetched based on the correct outcome of the branch. This penalty of discarding fetched instructions and fetching new ones, when the branch is mispredicted, is known as the branch misprediction penalty and it is very significant for most modern architectures. It is of the order of 5–10 cycles per branch instruction that is mispredicted. By reducing the branch instructions, the number of branches that get mispredicted automatically reduces.

2) It can better insert prefetches and effect other optimizations into the longer inner loop code. When the loop is unrolled, there are more memory instructions in the loop and also the memory stride (the distance between the memory accesses of an instruction in two consecutive iterations) is bigger. If a loop is unrolled four times, the memory stride goes up by four. This helps the prefetch to do a more effective job. When the memory stride increases, as long as it is less than the cache line size (which is architecture dependent), the prefetches become more effective. When the memory stride becomes greater than the cache line size the prefetches can hurt. Hence the loops should be unrolled such that the memory stride is lesser than the cache line size

whenever prefetch instructions are going to be generated for the loop and whenever possible.

3) Scalar replacement/recurrence elimination inserts copies at the loops to keep the value of a variable live in the next iteration (see, for example FIGS. 9, 10, and 11). These copies can be eliminated by unrolling the loop a certain number of times.

4) Longer sequences of non-branching instructions can achieve an overlap between instructions that have nothing to do with memory and those that do. This is known as instruction scheduling and explained below. While the access time between the processor and the cache is typically 1 to 5 cycles, the retrieval time from cache to memory is often on the order of 10 to 100 cycles. When the processor actually gets to the point where the data item is needed from memory, if the data is not in cache, it might take 100 processor cycles to fetch it from main memory. Where the compiler can optimize the longer inner loop code, it may only be necessary to wait for 20 cycles because 80 cycles worth of look up time is hidden or overlapped with the execution of other instructions.

Loop unrolling is integrated with other low level optimization phases, such as the prefetch insertion algorithm, register reassociation, and instruction scheduling. The new compiler yields significant performance improvements for some industry-standard performance benchmarks, for example on the SPEC92 and SPEC95 benchmarks on the Hewlett-Packard Company (Palo Alto, Calif.) PA-8000 processor.

The following discussion explains compiler operation in the context of a loop within an application program. Loops are readily recognized as a sequence of code that is iteratively executed some number of times. The sequence of such operations is predictable because the same set of operations is repeated for each iteration of the loop. It is common practice in an application program to maintain an index variable for each loop that is provided with an initial value, and that is incremented by a constant amount for each loop iteration until the index variable reaches a final value. The index variable is often used to address elements of arrays that correspond to a regular sequence of memory locations.

In the compiler, it has been found that the low level optimizer component of a compiler is in a good position to deduce the number of cycles required by a stretch of code that is repetitively executed and this information can be used to determine the optimal unroll factor. As discussed above, the concept of loop unrolling is not new, but use of smart unrolling is new. For example, FIG. 4 shows the loop of FIG. 3 after the loop has been unrolled four times. Thus, instead of executing the loop 100 times if N were 4, the loop is executed 25 times.

FIG. 5 shows the output code that is generated by the invention in contrast to the code generated by the prior art, as shown on FIG. 4. The replicated inner loops **333** are the same. Also, the compensation loop **323** is the same as the prior art compensation loop **155** of FIG. 4. However, the loop test at **301** and **159** of FIG. 4 now tests to exit if I>=5 rather than I>=4, as can be seen at **311** and **321** of FIG. 5. The effect is to ensure that the compensation loop is always executed at least once. This eliminates the need to test for the zero case (**303** in FIG. 4). This eliminates the branch instruction **303** on FIG. 4. As indicated above, the elimination of this branch instruction significantly increases the speed of the compiled code by reducing the number of branch instructions that get mispredicted.

It is also possible to put the compensation code in front of the main loop, as is shown on FIG. 6. Here the compensation loop **383** is in front of the repetitive unrolled loops **347**. In the general case, putting the compensation code before the main unrolled loop is less efficient than putting afterwards, because calculating the loop trip count requires a remainder operation which involves high latency divide operations.

However, if the unroll factor is a power of two, as in this case where the unroll factor is 4, the remainder calculation is a simple shift operation. Because unroll factors of 2, 4, or 8 are common, the compensation code can be placed in front in front of the unroll loop for negligible cost. As a practical matter, it is often advantageous to put the compensation loop in front of the unrolled loop to benefit from other optimizations such as register reassociation. When the compensation loop is placed before the unrolled loop, the variable that keeps track of the iteration count is not always needed after the unrolled loop. When the compensation loop is placed after the unrolled loop, this variable is always needed after the unrolled loop as there is an exposed use in the compensation loop. This exposed use can inhibit aggressive register reassociation. In the preferred embodiment, the architecture of the computer and the interactions with other optimizations dictate an unroll factor, and if it is a power of two, the compensation code is inserted in front of the unroll loop.

Another optimization technique that is part of the invention herein disclosed rearranges loops with early exits (which are henceforth referred to as WHILE loops). These loops are characterized by the fact that some of the inner loop code is done before the loop test, and some after the loop test as is shown on FIG. 7*a*. Here the loop has an exit branch **403** in the middle with inner loop operations **401** before it, and other inner loop operations **405** after it.

The optimization taught in the prior art for this loop is shown on FIG. 7*b*. Notice that the whole inner loop, including the exit instruction, is replicated **377** four times. This can be improved by converting the unrolled loop into a FOR loop with an exit condition of (unroll factor +1) as opposed to unroll factor (e.g. 5 instead of 4 in this case), as is shown on FIG. 8. This guarantees that the unrolled loop is exited before it would have to exit due to the WHILE condition. Because none of the branches at **377** on FIG. 7*b* are executed, the WHILE exit instruction can be removed, as is shown at **319** on FIG. 8. Thus, there is only one place that there is a WHILE loop exit, i.e. at **365**.

The technique herein disclosed ensures that the unrolled loop exits before it would take any of the WHILE loop exits, so that the WHILE test can be removed from the unrolled loop. It is necessary to ensure that the compensation code is always executed at least once.

The following discusses how the unroll factor interacts with the scalar replacement optimization. This is particularly important because the form of this optimization determines the unroll factor. Consider the loop shown on FIG. 9. Notice that the value of A[I] stored in the inner loop at **367** is loaded again two loop iterations later, when the same statement loads A[I-2] with a value of I which is incremented by 2. The idea behind scalar optimization which is well known in the prior art, is to save array values in temporary variables if they are accessed shortly within the next few iterations. Thus, the loop can be modified as is shown on FIG. 10.

Here, the array reference A[I-2] at **391** is replaced with T, and it is followed by two instructions at **393** and **395** which assign values to T and T1. Two scalar temporary variables are necessary because the value of the two most recent array values must be saved. The value of A[I-1] would have been stored in the previous iteration in T1 and that is going to be used in the next iteration. We move T1 to T and T will be accessed in the next iteration. Similarly T1, to which A[I] is assigned will be moved to T in the next iteration and used 2 iterations from now.

The instruction at **395** appears to make an indexed reference to the array A and that suggests that an array access must be made to get the number to put into T1, which would be a high latency operation and would lose all that was gained by the optimization. What actually happens is the optimizer recognizes that the value of A1 is stored two instructions earlier at **391**, and that A[I] is resident in a register which can be stored into T1 without accessing A[I]. As T1 is likely to be assigned to a register, this operation is a register to register instruction.

The foregoing illustrates how the various optimization techniques are interrelated allowing the loop unrolling optimizer to generate code which is clearly not optimal in itself but is optimized by other optimizers in the compiler. Initialization code is inserted at **388** to define initial values of T and T1.

FIG. **11** shows how such a loop can be unrolled according to the prior art if an unroll factor of three had been chosen. The prior art does not specify the selection of an unroll factor of three here, but a factor of three, or a multiple of three is optimal because it allows other parts of the optimizer to generate the code shown on FIG. **12**. The selection of an unroll factor here of three or a multiple of three is important because three values of the array must be kept A[I], A[I-1] and A[I-2] if the array references are to be avoided.

Three variables T, T1 and T2 are used, making it possible for other well known code optimization techniques to generate code, such as that shown on FIG. **12**. This eliminates shuttling the temporary data from T to T1. This is an example of where the nature of the code in the loop and its effect on scalar optimization forces a particular unroll factor. In general, one lists the various indexes in the loops and sorts them to notice the maximum distance between them.

In the above case the distance between the index I, and I-2, is 2. Adding one to this value computes a primary unroll factor, which is three in this case. This is an acceptable unroll factor. However, if it turned out to be very small, one might want to multiply it by a constant to get a larger unroll factor. Alternatively, if the primary unroll factor was very large, one might want to divide it by the loop increment of it was a number other than one. The reasons for selecting a particular unroll factor are discussed below.

Determining the unroll factor.

Classically the prior art uses a standard unroll factor for all loops. Typically the number used is four. In the invention, the unroll factor is calculated for each loop depending on various factors. At one extreme some loops are not unrolled at all, and other loops are unrolled eight or even more times. The disadvantages of picking too large an unroll factor are:

1. All the loop instructions need not fit into the instruction cache leading to a lot of I-cache misses.
2. The higher the unroll factor, the higher the memory stride of memory instructions across iterations. If the memory stride exceeds cache line size, the effectiveness of prefetch decreases.
3. The resulting code is longer. Usually an upper bound must be chosen, an unroll count of 1000 is not likely to be a good idea since the compile time can go up significantly. Also excessive unrolling can adversely affect other optimizations which have bounds on the number of transformations they can make.

On the other hand if a small unroll factor is chosen the following problems can occur:

1. Much more time is spent executing the high latency branch instruction which closes the loop.
2. The short inner loop provides many fewer opportunities for optimization than longer inner loops. Where the inner loop has high latency instructions, the compiler can often have them execute in parallel with low latency time instructions. This may not be possible in very short loops.

One must keep in mind that the compiler is compiling loops that range from single instruction inner loops to loops that have scores or even hundreds of instructions, and so the compiler must compile code balance these considerations to achieve good unroll factors. To determine the unroll factor, the compiler considers the following in decreasing order of importance:

1. There is a maximum value of the unroll factor (which in the preferred embodiment of the invention is eight).
2. The number of instructions in the unrolled loop must not exceed a specific limit. This provides another upper bound to the unroll factor.
3. If there are references to previous indexed contents of the array such as was shown in FIGS. **10** through **12**, an unroll factor suggested by this analysis (or a multiple of it) should be used.
4. If prefetch instructions are being generated (this is known based on a user defined flag), then try to pick an unroll factor that keeps the value of the strides of array references within the loop below the cache line size.
5. If the trip count is a constant known at compile time, then an unroll factor that eliminates the need for a compensation code loop should be selected. Typically this would be an unroll factor of 2, 4 or 8, although other numbers such as 3 or 5 might be possible.
6. If there is profile information, use that. If the profile informations says that the loop iterates on an average k̀ times, if k̀ is smaller than the maximum value of the unroll factor as dictated by the previous steps, use k̀, else use the maximum value of the unroll factor.
7. If there are high latency operations within the loop such as divide and square root operations, use an unroll factor that will enhance the maximum overlap of these instructions. For instance, if the architecture has two divide units and the loop has a single divide instruction, the loop should be unrolled an even number of times so that both the divide units can be kept simultaneously busy.

The algorithm that computes the unroll factor tries to compute an optimal and acceptable unroll factor. The cost of a nonoptimal unroll factor is slower run time code. As discussed above, the algorithm is sensitive to profile data, number of instructions in the loop, architecture features like functional units and cache line size, interactions with other optimizations and constant trip counts.

Attention is directed to FIG. **13**, which shows how the optimization algorithms presented here are implemented. At **201** the unroll factor is determined as described above. Next, at **203**, a check is made for the special case where the trip count is known at compile time and is a multiple of the unroll factor. In this case, the unrolled loop code is generated from the original loop code, any middle exits are removed leaving only the final exit, and the unrolled code is output at **242**. Because this is an unrolled loop which needs no compensation code, none is output. The other exit occurs at **203** where the trip count is not known at compile time, or the trip counts and unroll factor are such that compensation code must be generated. Control goes to **205** where the unrolled code is generated. For non-WHILE loops, the middle exits are removed leaving only the final exit. For a WHILE loop, the final exit is moved to the end of the loop (**226**) instead of the middle and all other exits removed. At this time a determination (at **229**) is made using the unroll factor to determine if the compensation code should be output before or after the unrolled loop code. If it should be after, control goes to **244**, otherwise control goes to **246**. All of these three control paths then meet at **999**, terminating the unrolling optimization.

FIG. **14** is a block schematic diagram of a compiler for a programmable machine in accordance with the invention. The compiler of FIG. **2*b*** shows a loop unrolling module **30**. The preferred embodiment of the invention provides a loop unrolling module that is placed within the compiler as shown in FIG. **2*b***. As shown in FIG. **14**, the compiler comprises an analysis module **1301** for analyzing and unrolling loops within source applications. An optimizer module **1302** determines an optimum unroll factor in response to the analysis module. An unroll module **1303** generates an unrolled loop having said optimum unroll facto, while a compensation module **1304** generates and places any compensation code as required as a result of loop unroll optimization.

Although the invention is described herein with reference to the preferred embodiment, one skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention. Accordingly, the invention should only be limited by the claims included below.

We claim:

1. In a programmable machine, a compiler comprising:
   - an analysis module for analyzing and unrolling loops within source applications;
   - an optimizer module for determining an optimum unroll factor in response to said analysis module;
   - an unroll module for generating an unrolled loop having said optimum unroll factor; and
   - a compensation module for generating and placing any compensation code as required as a result of loop unroll optimization,
   - wherein said compensation module performs a placement calculation to determine whether to put said compensation code before the unrolled loop or after the unrolled loop.

2. The compiler of claim **1**, wherein said compensation module ensures that said compensation code is executed at least once when said unrolled loop is executed.

3. The compiler of claim **1**, wherein said unroll factor is responsive to a number of instructions in the loop.

4. The compiler of claim **1**, wherein said unroll factor is responsive to resource usage within the loop.

5. The compiler of claim **1**, wherein said unroll factor is responsive to prefetch distance of memory references within the loop.

6. The compiler of claim **1**, wherein said unroll factor is responsive to profile information collected from previous executions of compiled code.

7. The compiler of claim **1**, wherein said unroll factor is responsive to recurrence of memory references within the loop.

8. The compiler of claim **1**, wherein said unroll factor is responsive to the number of instructions in the loop.

9. The compiler of claim **1**, wherein a trip count for the loop is known at compile time, and wherein said optimizer module determines an unroll factor that executes the compensation code zero times and suppresses the generation of said compensation code.

10. The compiler of claim **1**, wherein said placement calculation is responsive to the unroll factor computed for the loop.

11. The compiler of claim **1**, wherein said loop is a loop having an early exit.

12. The compiler of claim **1**, wherein loop unrolling is integrated with other low-level optimization phases.

13. The compiler of claim **12**, wherein said other low-level optimization phases include any of prefetch instruction insertion, register reassociation, and instruction scheduling.

14. A method for unrolling loops, comprising the steps of:
   - determining a unroll factor;
   - generating an unrolled loop which always exits leaving a remaining trip count of at least one;
   - generating compensation code;
   - determining whether the compensation code should be placed before the unrolled loop or after the unrolled loop; and
   - determining if the trip count is a power of two; and if it is placing the compensation code before the unrolled loop.

15. The method of claim **14**, wherein the loop to be unrolled is a loop having an early exit.

16. The method of claim **15**, wherein the loop having an early exit after unrolling is transformed into a loop having an exit at its end, and from which all intermediate exits have been removed.

* * * * *